Daml SDK Documentation

Digital Asset

Version : 2.7.3

Table of contents

Bibliography **1843**

Chapter 1

Canton References

1.1 An Introduction To Multi-Party Applications and Daml

1.1.1 Multi-Party Applications

Multi-party applications, and multi-party application platforms like Daml, solve problems that were near

why multi-party applications matter
what a multi-party application is
important concepts in multi-party applications
key architectural concepts in Daml
a transfer example using Daml

1.1.2 Why Do Multi-Party Applications Matter?

Have you ever wondered why bank transfers, stock purchases or healthcare claims take days to process? Given our technological advances, including the speed of networks, you might expect these transactions to take less than a second to complete. An inefficient protocol like email takes only a few seconds to send and receive, while these important business workflows take days or weeks.

What delays these transactions? The processes in question all involve multiple organizations that each keep their own records, resulting in data silos. The processes to ensure consistency between those data silos are complex and slow. When inconsistencies arise, the correction processes (sometimes referred to as reconciliation) are expensive, time-consuming and often require human intervention to determine why two parties have differing views of the result of a business interaction. There are a myriad of reasons for these discrepancies, including differences in data models and error handling logic, inconsistent business process implementations and system faults.

Here's a deeper look at the problem via the example of a transfer of $100 from Alice's account in Bank A to Bob's account in Bank B. (Money is an easily understood example of an asset transferred between parties. The same problems occur in other industries with other assets, for example, healthcare claims, invoices or orders.) Money cannot simply appear or disappear during a transfer. The banks need to ensure that at some point in time, T_0, $100 are in Alice's account, and at the next point in time, T_1, those $100 are gone from Alice's account and present in Bob's account – but at no point are the $100 present in both accounts or in neither account.

In legacy systems, each bank keeps track of cash holdings independently of the other banks. Each bank stores data in its own private database. Each bank performs its own processes to validate, secure, modify and regulate the workflows that transfer money. The coordination between multiple banks is highly complex. The banks have an obligation to limit their counterparty risk - the probability that the other party in the transaction may not fulfill its part of the deal and may default on the contractual obligations.

Today's common, albeit highly inefficient and costly, solution for a bank account transfer involves the following steps:

1. Bank A sends a message to Bank B via a messaging standard and provider like SWIFT or SEPA.
2. Bank A and Bank B determine a settlement plan, possibly including several intermediaries. Gaining an agreement on the settlement plan is time-consuming and often includes additional fees.
3. The settlement process entails (i) debiting $100 from Alice's account at Bank A, (ii) crediting the commercial account at Bank B, and (iii) once Bank B has the money, crediting Bob's account at Bank B.

In order to make this process atomic (that is, to make it take place between a point T_0 and a point T_1) banks discretize time into business days. On day T_0 the instruction is made and a settlement plan is created. Outside of business hours between day T_0 and day T_1, the plan is executed through end of day netting and settlement processes. In a sense, banks agree to stop time outside of business hours.

If intermediaries are involved, the process is more complex. Cross-border payments or currency conversion add yet more complexity. The resulting process is costly and takes days. During this multi-day process the $100 is locked within the system where it is useless to both Alice and Bob. Delays are common, and if there are problems reconciliation is hugely expensive. Consolidating through centralized intermediaries introduces systemic risk, including the risk of unauthorized disclosure and privacy breaches - and with that risk comes increased latency. Banks insist on this approach, despite the downsides, to reduce counterparty risk and to comply with regulations. At every point in time, ownership of the money is completely clear. (To learn more about cash transfers in traditional banking systems, read this accessible writeup on international money transfers.)

Services like PayPal, Klarna and credit cards, which provide an experience of instant payments internationally, do this by accepting the counterparty risk or acting as banks themselves. If a shop accepts credit cards and you pay with a credit card, both you and the shop have an account with the credit card company. When you purchase, the credit card company can instantly debit $100 from your account and credit $100 to the shop's account because it is as if both Alice and Bob are using accounts at the same bank – the bank is certain that Alice has $100 in her account and can execute a simple transaction that deducts $100 from Alice's account and adds $100 to Bob's.

Wouldn't it be great if a system existed that allowed multiple parties to transact with each other with the same immediacy and consistency guarantees a single organization can achieve on a database while each kept sovereignty and privacy of their data? That's Daml!

Daml is a platform and framework for building real-time multi-party systems, enabling organizations to deliver the experiences modern users expect without assuming counterparty risk or the expense of reconciliation. The sections below describe how Daml achieves this, including the architectural concepts and considerations necessary to build and deploy a solution with Daml effectively.

1.1.3 What Is a Multi-Party Application?

A multi-party application is one in which data, and the rules and workflows that govern the data, are shared between two or more parties without any party having to give up sovereignty or any single party (including the application provider) being able to control or override the agreed rules of the system. A party could be a company, a department within a company, an organization, an individual or a person. The specific definition of a party will be unique to the application and the domain of that application.

A well-designed multi-party application provides several benefits:

a clean, consistent view of all data managed by the application across all parties
consistent, connected, and efficient processes between all parties involved in the application
privacy controls over portions of the shared data, such that each party sees only the data that it is explicitly entitled to view and/or modify
individual party ownership of and responsibility for sensitive data

In most cases, no single party can view all of the data within a multi-party application.

Multi-party applications solve complex operational processes while keeping data clean and consistent, thereby eliminating isolated, disconnected and inefficient processes that often require expensive reconciliation. Multi-party applications manage the relationships, agreements and transactions between parties, providing consistent real-time views of all data.

Multi-party solutions utilize distributed ledger (blockchain) technology to ensure each party has an immutable, consistent view of the shared data and business processes that govern the data. By providing a consistent view of data with all counterparties, a multi-party application removes friction, cost, and risk within a joint business process. A distributed ledger protects against a malicious participant in the network, attempting to write or overwrite data to the detriment of other parties.

1.1.4 Important Concepts in Multi-Party Applications

For a multi-party application to fully deliver its value, the following conditions must be met:

Multiple involved parties have data sovereignty – that is, they keep their data within their own systems and require strong guarantees that no external party can access or modify that data outside of pre-agreed rules. Shared state and rules are codified into an executable schema that determines what data can move between parties, who can read that data, and how that data is manipulated. Processes happen in real time as there is no additional reconciliation or manual processing required between organizations.

For each individual party to gain the full benefits of a multi-party system, it should:

Integrate the application - Bank A must treat the multi-party infrastructure as the golden source of truth for payment information and integrate it as such with the rest of their infrastructure. Otherwise they are merely trading inter-bank reconciliation for intra-bank reconciliation.
Utilize composability by building advanced systems that rely on the base-level multi-party agreements. For example, a healthcare claim application should be built using the payment solution. Integrating one multi-party application with another preserves all the properties of each across both applications. In this example, the patient privacy requirements of a health claims application are retained, as are the financial guarantees of the payment application. Without composability, multi-party applications become bigger si-

los and you end up reconciling the healthcare claims multi-party application with the payments multi-party application.

Smart contracts, distributed ledgers, and blockchains are commonly used to build and deliver multi-party applications. A smart contract codifies the terms of the agreement between parties, including the rights and obligations of each party, directly written into lines of code. The code controls the execution, and transactions are trackable and irreversible. In a multi-party application, the smart contract defines the data workflow through actions taken by the parties involved.

Distributed ledgers and blockchains provide consensus between the parties, with a cryptographic audit trail maintained and validated by the system. Within multi-party solutions, the distributed ledger ensures no one party can unilaterally change the system's state and protects data sovereignty, while the distributed ledger synchronizes the nodes securely in real time.

1.1.5 Key Architectural Concepts in Daml

Daml comprises two layers necessary for building multi-party applications: the Daml smart contract language and the Canton blockchain and protocol.

The Daml language is a smart contract language for multi-party applications. Conceptually, Daml is similar to the Structured Query Language (SQL) used in traditional database systems, describing the data schema and rules for manipulating the data.

The Daml language:

defines the shared state between the parties, including process permissions and data ownership
defines workflows, execution policies, and read/write permissions
enables developers to build rich transactions that codify strict business rules
defines the APIs through which multi-party applications can talk to each other and compose

The Daml code that collectively makes up the data schema and rules for an application is called a Daml model. Increasingly sophisticated and valuable solutions are composed from existing Daml models, enabling a rich ecosystem that accelerates application development.

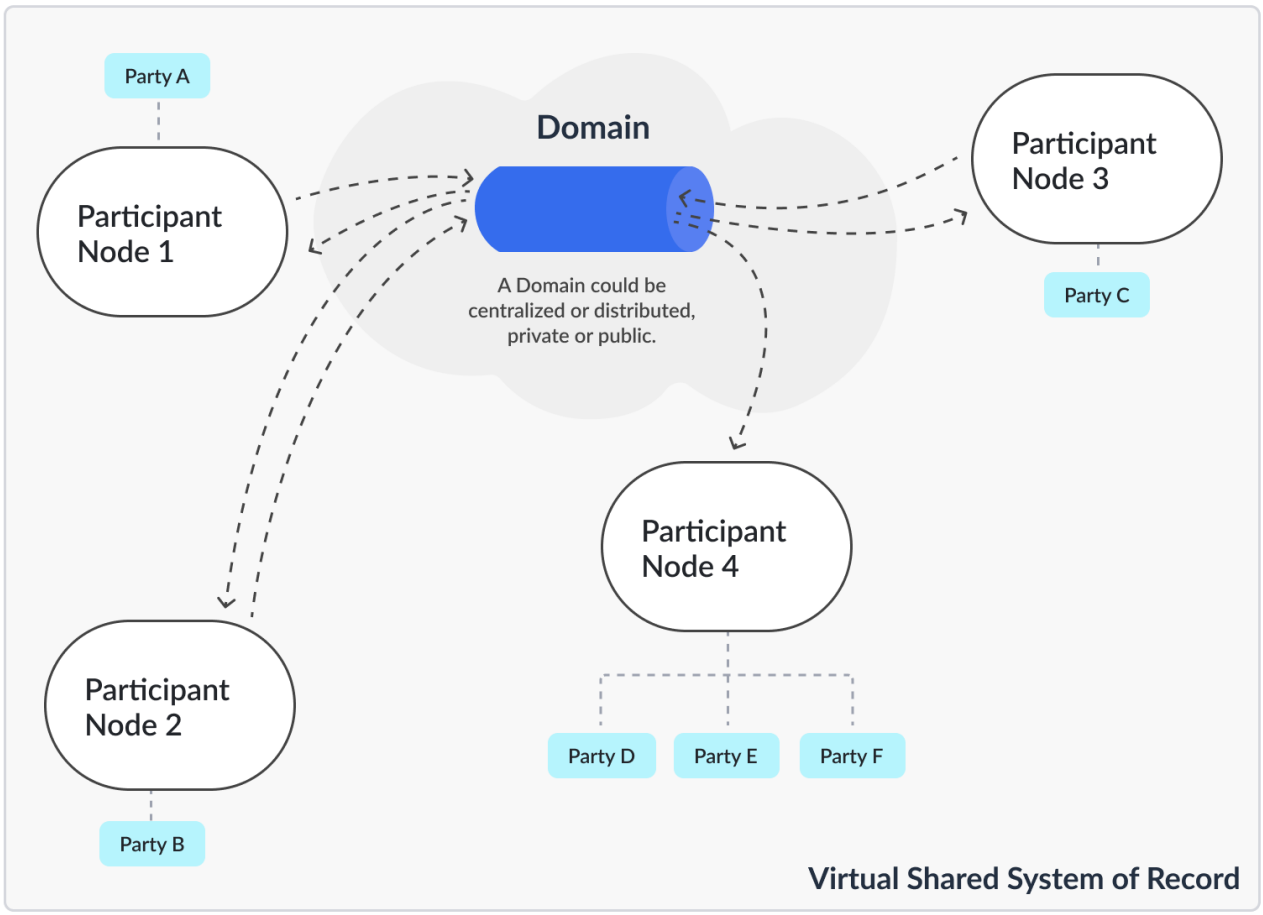
Using the Daml language, developers define the schema for a virtual shared system of record (VSSR). A VSSR is the combined data from all parties involved in the application. The Canton protocol ensures that each party gets a unique view into the VSSR, which is their projection of the full system.

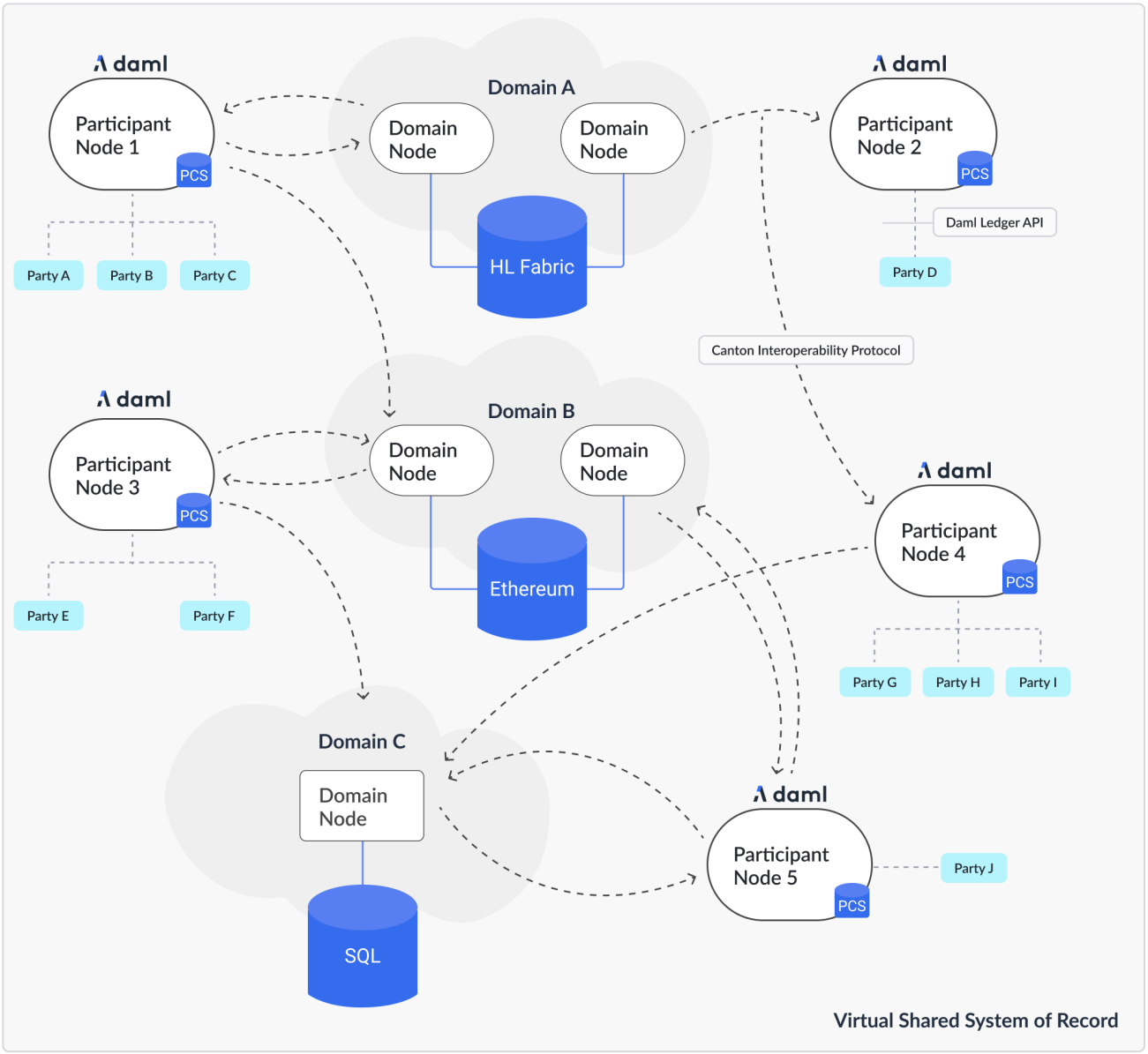
In the execution model for Canton, each party of the application is hosted on a Participant Node (Diagram 1). The Participant Node stores the party's unique projection and history of the shared system of record. Participant Nodes synchronize by running a consensus protocol (the Canton Protocol) between them. The protocol is executed by sending encrypted messages through Domains, which route messages and offer guaranteed delivery and order consistency. Domains are also units of access control and availability, meaning an application can be additionally protected from interference by other applications or malicious actors by synchronizing it only through a given domain, and restricting which participants can connect to it.

Diagram 1:

In a composed solution, each domain is a sub-network. A Participant Node connects to one or more Domains, enabling transactions that span Domains (Diagram 2).

Diagram 2:


Party A
Participant Node 1
Domain
A Domain could be centralized or distributed, private or public.
Participant Node 3
Party C
Participant Node 4
Participant Node 2
Party D
Party E
Party F
Party B
Virtual Shared System of Record


daml
Participant Node 1
PCS
Party A
Party B
Party C
Domain A
Domain Node
Domain Node
HL Fabric
daml
Participant Node 2
PCS
Daml Ledger API
Party D
Canton Interoperability Protocol
daml
Participant Node 3
PCS
Party E
Party F
Domain B
Domain Node
Domain Node
Ethereum
daml
Participant Node 4
PCS
Party G
Party H
Party I
Domain C
Domain Node
SQL
daml
Participant Node 5
PCS
Party J
Virtual Shared System of Record

1.1.6 Transfer Example Using Daml

Consider the transfer example described above with Alice and Bob. Using Daml, the process looks like this:

1. Alice logs into her online banking at Bank A and enters a transfer to Bob at Bank B.
2. The online banking backend creates a transaction that deducts $100 from Alice's account and creates a transfer to Bob at Bank B.
3. When Bank B accepts the transfer, Bank A credits $100 to Bank B's account at Bank A and Bank B simultaneously credits Bob's account by $100.
4. Bob's online banking interfaces with the Daml Ledger and can see the incoming funds in real time.

At every point, ownership of the $100 is completely clear and all systems are fully consistent.

1.1.7 Next Steps

The suggested next steps are:

> Learn about the Daml language and the Daml Ledger Model. *Writing Daml* will introduce you to the basics of a Daml contract, the Daml Ledger model, and the core features of the Daml language. You'll notice that testing your contracts, including *testing for failures*, is presented very early in this introduction. We strongly recommend that you write tests as part of the initial development of every Daml project.
> Learn about operating a Daml application with the *Ledger Administration Introduction*.

1.2 System Requirements

Unless otherwise stated, all Daml runtime components require the following dependencies:

1. For development, an x86-compatible system running a modern Linux, Windows, or MacOS operating system. For a production deployment, an x86-compatible system running a modern Linux operating system.
2. Java 11 or greater.
3. An RDBMS system,

1. Either PostgreSQL 11.17 or greater.
2. Or Oracle Database 19.11 or greater.

4. JDBC drivers compatible with the chosen RDBMS.

Daml is tested using the following specific dependencies in default installations.

1. Operating Systems:

1. Ubuntu 20.04 for development. Ubuntu 20.04 and Debian 11 is also tested for production use.
2. Windows Server 2016
3. MacOS 10.15 Catalina

2. Eclipse Adoptium version 11 for Java.
3. PostgreSQL 11.17
4. Oracle Database 19.11

In terms of hardware requirements, minimal deployments running simple Daml applications are regularly tested with as little as 2 GB of memory and access to a single, shared vCPU.

1.2.1 Feature/Component System Requirements

1. The JavaScript Client Libraries are tested on Node 14.18.3. with typescript compiler 4.5.4. Versions greater or equal to these are recommended.

1.3 Installing the SDK

1.3.1 Install the Dependencies

The Daml SDK currently runs on Windows, macOS and Linux.

You need to install:

1. Visual Studio Code.
2. JDK 11 or greater. If you don't already have a JDK installed, try Eclipse Adoptium.
   As part of the installation process you may need to set up the `JAVA_HOME` variable. You can find instructions for this process on *Windows,macOS, and Linux here*.

1.3.2 Choose Daml Enterprise or Daml Open Source

Daml comes in two variants: Daml Enterprise or Daml Open Source. Both include the best in class SDK, Canton and all of the components that you need to write and deploy multi-party applications in production, but they differ in terms of enterprise and non-functional capabilities:

Capability	Enterprise	Open Source
Sub-Transaction Privacy	Yes	Yes
Transaction Processing	Parallel (fast)	Sequential (slow)
High Availability	Yes	No
Horizontal scalability	Yes	No
Ledger Pruning	Yes	No
Local contract store in PostgreSQL	Yes	Yes
Local contract store in Oracle	Yes	No
PostgreSQL driver	Yes	Yes
Oracle driver	Yes	No
Besu driver	Yes	No
Fabric driver	Yes	No
Profiler	Yes	No
Non-repudiation Middleware	Yes (early access)	No

1.3.3 Install Daml Open Source SDK

1.3.3.1 Windows 10

Download and run the installer, which will install Daml and set up the PATH variable for you.

1.3.3.2 Mac and Linux

Open a terminal and run:

```
curl -sSL https://get.daml.com/ | sh
```

The installer will setup the `PATH` variable for you. In order for it to take effect, you will have to log out and log in again.

If the `daml` command is not available in your terminal after logging out and logging in again, you need to set the `PATH` environment variable manually. You can find instructions on how to do this *here*.

1.3.4 Install Daml Enterprise

If you have a license for Daml Enterprise, you can install it as follows:

On Windows, download the installer from Artifactory instead of Github releases.
On Linux and MacOS, download the corresponding tarball, extract it and run `./install.sh`.
Afterwards, modify the *global daml-config.yaml* and add an entry with your Artifactory API key.
The API key can be found in your Artifactory user profile.

```
artifactory-api-key: YOUR_API_KEY
```

This will be used by the assistant to download other versions automatically from artifactory.

If you already have an existing installation, you only need to add this entry to `daml-config.yaml`.
To overwrite a previously installed version with the corresponding Daml Enterprise version, use `daml install --force VERSION`.

1.3.5 Download Manually

If you want to verify the SDK download for security purposes before installing, you can look at *our detailed instructions for manual download and installation*.

1.3.6 Next Steps

Follow the *getting started guide*.
Use `daml --help` to see all the commands that the Daml assistant (`daml`) provides.
If you run into any other problems, you can use the *support page* to get in touch with us.

1.4 Setting JAVA_HOME and PATH Variables

1.4.1 Windows

To set up `JAVA_HOME` and `PATH` variables on Windows:

1.4.1.1 Set the JAVA_HOME Variable

1. Search for Advanced System Settings (open `Search`, type `advanced system settings` and hit `Enter`).
2. Find the `Advanced` tab and click `Environment Variables`.
3. Click `New` in the `System variables` section (if you want to set `JAVA_HOME` system wide) or in the `User variables` section (if you want to set `JAVA_HOME` for a single user). This will open a modal window for `Variable name`.
4. In the `Variable name` window type `JAVA_HOME`, and for the `Variable value` set the path to the JDK installation.
5. Click OK in the `Variable name` window.
6. Click OK in the tab and click Apply to apply the changes.

1.4.1.2 Set the PATH Variable

The `PATH` variable is automatically set by the Windows installer .

1.4.2 Mac OS

First, determine whether you are running Bash or zsh. Open a Terminal and run:

```
echo $SHELL
```

This should return either `/bin/bash`, in which case you are running Bash, or `/bin/zsh`, in which case you are running zsh.

If you get any other output, you have a non-standard setup. If you're not sure how to set up environment variables in your setup, ask on the Daml forum and we will be happy to help.

Open a terminal and run the following commands. Copy/paste one line at a time if possible. None of these should produce any output on success.

To set the variables in **bash**:

```
echo 'export JAVA_HOME="$(/usr/libexec/java_home)"' >> ~/.bash_profile
echo 'export PATH="$HOME/.daml/bin:$PATH"' >> ~/.bash_profile
```

To set the variables in **zsh**:

```
echo 'export JAVA_HOME="$(/usr/libexec/java_home)"' >> ~/.zprofile
echo 'export PATH="$HOME/.daml/bin:$PATH"' >> ~/.zprofile
```

For both shells, the above will update the configuration for future, newly opened terminals, but will not affect any existing one.

To test the configuration of `JAVA_HOME` (on either shell), open a new terminal and run:

```
echo $JAVA_HOME
```

You should see the path to the JDK installation, which is something like `/Library/Java/JavaVirtualMachines/jdk_version_number/Contents/Home`.

Next, please verify the `PATH` variable by running (again, on either shell):

```
daml version
```

You should see the header `SDK versions:` followed by a list of installed (or available) SDK versions (possibly a list of just one if you just installed).

If you do not see the expected outputs, contact us on the Daml forum and we will be happy to help.

1.4.3 Linux

To set up `JAVA_HOME` and `PATH` variables on Linux for `bash`:

1.4.3.1 Set the JAVA_HOME Variable

Java is typically installed in a folder like `/usr/lib/jvm/java-version`. Before running the following command make sure to change the `java-version` with the actual folder found on your computer:

```
echo "export JAVA_HOME=/usr/lib/jvm/java-version" >> ~/.bash_profile
```

1.4.3.2 Set the PATH Variable

The installer will ask to set the `PATH` variable for you. If you want to set the `PATH` variable manually instead, run the following command:

```
echo 'export PATH="$HOME/.daml/bin:$PATH"' >> ~/.bash_profile
```

1.4.3.3 Verify the Changes

In order for the changes to take effect you will need to restart your computer. After the restart, verify that everything was set up correctly using the following steps:

Verify the JAVA_HOME variable by running:

```
echo $JAVA_HOME
```

You should see the path you gave for the JDK installation, which is something like `/usr/lib/jvm/java-version`.

Then verify the PATH variable by running:

```
echo $PATH
```

You should see a series of paths which includes the path to the SDK, which is something like `/home/your_username/.daml/bin`.

1.5 Manually Installing the SDK

If you require a higher level of security, you can instead install the Daml SDK by manually downloading the compressed tarball, verifying its signature, extracting it and manually running the install script.

Note that the Windows installer is already signed (within the binary itself), and that signature is checked by Windows before starting it. Nevertheless, you can still follow the steps below to check its external signature file.

To do that:

1. Go to https://github.com/digital-asset/daml/releases. Confirm your browser sees a valid certificate for the github.com domain.
2. Download the artifact (*Assets* section, after the release notes) for your platform as well as the corresponding signature file. For example, if you are on macOS and want to install the latest release (2.0.0 at the time of writing), you would download the files `daml-sdk-2.0.0-macos.tar.gz` and `daml-sdk-2.0.0-macos.tar.gz.asc`. Note that for Windows you can choose between the tarball (ends in `.tar.gz`), which follows the same instructions as the Linux and macOS ones (but assumes you have a number of typical Unix tools installed), or the installer, which ends with `.exe`. Regardless, the steps to verify the signature are the same.
3. To verify the signature, you need to have `gpg` installed (see https://gnupg.org for more information on that) and the Digital Asset Security Public Key imported into your keychain. Once you have `gpg` installed, you can import the key by running:

   ```
   gpg --keyserver hkp://pgp.mit.edu --search
   ↪F26D8A0AADF666CCB28F2AB1650EC3253B6A8FF5
   ```

   This should come back with a key belonging to `Digital Asset Holdings, LLC <security@digitalasset.com>`, created on 2023-01-10 and expiring on 2025-01-09. If any of those details are different, something is wrong. In that case please contact Digital Asset immediately.
   Alternatively, if keyservers do not work for you (we are having a bit of trouble getting them to work reliably for us), you can find the full public key at the bottom of this page.
4. Once the key is imported, you can ask `gpg` to verify that the file you have downloaded has indeed been signed by that key. Continuing with our example of 2.0.0 on macOS, you should have both files in the current directory and run:

   ```
   gpg --verify daml-sdk-2.0.0-macos.tar.gz.asc
   ```

   and that should give you a result that looks like:

   ```
   gpg: assuming signed data in 'daml-sdk-2.0.0-macos.tar.gz'
   gpg: Signature made Wed Aug 12 13:30:49 2020 CEST
   gpg:                using RSA key CADC3D1E3B5C4C5F94A65D78A7BF65AAADBBC494
   gpg: Good signature from "Digital Asset Holdings, LLC <security@digitalasset.
   ↪com>" [unknown]
   gpg: WARNING: This key is not certified with a trusted signature!
   gpg:          There is no indication that the signature belongs to the owner.
   Primary key fingerprint: F26D 8A0A ADF6 66CC B28F  2AB1 650E C325 3B6A 8FF5
        Subkey fingerprint: CADC 3D1E 3B5C 4C5F 94A6  5D78 A7BF 65AA ADBB C494
   ```

   Note: This warning means you have not told gnupg that you trust this key actually belongs to Digital Asset. The `[unknown]` tag next to the key has the same meaning: `gpg` relies on a web of trust, and you have not told it how far you trust this key. Nevertheless, at this point you have verified that this is indeed the key that has been used to sign the archive.

5. The next step is to extract the tarball and run the install script (unless you chose the Windows installer, in which case the next step is to double-click it):

```
tar xzf daml-sdk-2.0.0-macos.tar.gz
cd sdk-2.0.0
./install.sh
```

6. Just like for the more automated install procedure, you may want to add `~/.daml/bin` to your `$PATH`.

To import the public key directly without relying on a keyserver, you can copy-paste the following Bash command:

```
gpg --import < <(cat <<EOF
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBGO9khIBEAC/D5WTgMJQGQso1JfN5RTq6YiCBwJ+L84YfKCPUo1yW7/RQHNZ
+5rYUQpGf1K5KCIhHtJeQyANzPy9KWnhDX6lIaoau6Dg9JK3SwNv20jDyCzZOjNW
Gfajy7xVTWXmYM/us8/A5kJN4pwEGIUL73n2uOtOzhpJ6TGLujNKB5EfGUO1L2Jr
v9BGx2ghv+dbdR3kPX6SYuj7U+tDvoaqJB8729kL14grpBqYy2YhF5eoLyvBaE9x
brDydUCu5t2Xpr7yI7xGOhUSn2ygoP3e9YSjOhowj5U5oFtTGxvqSf7xd9gkFaZY
uA58X3su0nxZ/9nbvb2RJPKtlUeOJS8pggXVSSGrHfWw3Bnu2G1pQNO+MYCS0Cu/
gMxQTnJ4itUNoFb3c9dSnB/VXWxsvlK3F+EdFg9HLNiStJVxPhPwgTo138ohTI1H
4eGdXpRPZSKNXGRRtWdbEseYBSDBzR0ulAn5TDXFDFjjJ5u7KJfdN7p9YaXWkXpB
+hvsiWJuvUDxTGlQE02PQjyN5vzj1NaU7CRRLvOYSstsOyTmuYg/xxvqA9XbPdti
g9AtaeYSjRzq7OBq79FhcmKDOfh7Zc07RRXHy2xTdvw+Iy5HEjk0fYFz+1Gtp78U
0iTv8tdqyh8dPvmuF7UbGWMJEMMD5d2goEw2ZnkqmLPFK5jq8qAshaQw9wARAQAB
tDdEaWdpdGFsIEFzc2V0IEhvbGRpbmdzLCBMTEMgPHNlY3VyaXR5QGRpZ2l0YWxh
c3NldC5jb20+iQJOBBMBCAA4FiEE8m2KCq32Zsyyjyqx ZQ7DJTtqj/UFAmO9khIC
GwMFCwkIBwIGFQoJCAsCBBYCAwECHgECF4AACgkQZQ7DJTtqj/WMbg/+K0Mte9y+
fCaWxFctfUbtd/JZBzpSCVMLN7PjZYZ50SwN/CqILUTFzzVLIx7uj/CyH/e1IV2O
RR7mWFTSADmkdrM45RBCvDs2UEIl3Rpsg/4iRpCZo01YQL9Y1XyUid8F3cQYmwPk
4YMY+tqqEhObAq0ngrGWiEWMUixbbRVqlPvRZDMeUNGdvmSOCs9LZLEnE9m4g2Kn
lNKddfLZ+sHaq2bfOiB+mZECX6wTusjqQWeJPRdflVWwMxZ7IkG9YoQHGlg8fTMd
3NqPE9OHOQiZhN4MbY6QZ70WexUNab8Pzf1Co4sSGhywVI3JibcqCNIbHW21+1py
OItJvdMxeSscOde2Fm5Dqmhf8UE+xgvPXa5xA5Yf40AqwuKt7boGsMf09Lf7zitX
5Zzl81saIPVC4OcM51t+sNDP6uJIynP5Dp1fxaIlb8gcQDqyWB/REr0vY1pRf/61
M8+jfUP3RJMbX/tUiCxEG+1uDSGTqj2Ac4TqiXfFKpg+TdEzNFj9VtrzTJT/tIgj
QlrKM9P9iB/JrNtqgeYrhaBZSpVKx4J7LNeIGdVJvRVzlW3tvCsTIT/lp/iJ1YjI
FCdb76leR/PgQNdk4wyU4JLXOYueEPAbyiBqQwgmOoT8GpY1PP4dsFfu7MoV0Cq7
//q+uwegRr5lLV6LwSBuFd1hqQ9ZdjAmmRi5Ag0EY72SEgEQAKP+D3bVJPC6sxSj
q/3UH9hixNhcmG61w6X1uW0x5jMMYN72ilnDLbgsgA3qEyZ8G/i34nUU4K/WZkWg
nJ59lOPIVf05yzEnesS6hbHXUzd6ayeWhPUzwxLBPy3yJUw7IRkFF9P9AMBaraAp
27ZuWy40Ta8bVKc9DgEeWuesyFAqs74W7cRfGm0SCAp8R3I+Syoj66+jpXYJ7sFt
eW4ITqrQcj64jBtGB8kQOe8JvC4COudXJ1BpKjExxIQlSK729tz0vsi+hzQfac/1
m3j082sH89ZU8y4GQpjWo6YyEzIxKBgoEogD0CvYOeJ9nK1Uv3pVFKyC1KdysQ+h
v+9V3zQoOaGF6115cIwQU1ewISUkiCOHzMYkrEXsbBOJlCmomuLnjMhsXht5tV4e
c8axn6QM7qRfSR/3R0RZwdAca0oZBN4ZOokUuZnR7/FxyiOhKilGW5iX+0m1VvKH
BImFM/VmCXw4hzcWZUZa5K6Ebpeg7zWN3a1kXZ+Kb2glqWYT5Pq3d1m+RtJOiuyn
uyr1BnX6OvjTNWTmKPqO8x223dZpNGdK6sfUUeZ67OokI/l2dALOuZRcuCLK32LB
uJmk/dLt4Bjem9ITFt2ECb1+RTa1aWOm8uS7BKUiDGedW6239h3HebdVenip1voY
3EdwpiQxgsCD3g2Sbzj9M5UGOsWzABEBAAGJAjwEGAEIACYCGwwWIQTybYoKrfZm
zLKPKrFlDsMlO2qP9QUCY72SxAUJA8JnsgAKCRBlDsMlO2qP9dfyD/9O76RZYI6w
8xIEOoK/cw//4IA0bbN/vC2tn5l1zUba6TrXhCYKr96//YJS9Fd239Gf4kC7AEbS
yf4ARLbezjtOVG33GlfrEFHfghMKhpjMQgb68NFw5U2eLMFc7BB/Fu4vSHqCMZ3I
ajM/465kq+jLxTNiuI14MFs1OLGD5WbAo9VEzBUbi3mK/CB4xv2UEd2y6ZAZuCXO
P2+Pr2P7W94ECu/N0dhnitkAirgXrS3nZSduLpjK/SkUzvdY642GHwy0i3M20Ztr
p7o1Uu7ztlD9yDUbksMyhskG7I+k2NGLAwz/CG91GRrYdUpoWsPlU5XLyxjHCmSC
```

(continues on next page)

(continued from previous page)

```
q97qiRSKlGO3LbIiTRatrv+4fcdntN0EM/nJefdtKS8+qZqkPMGqURlDJcPnIpHk
jGccrEJz4aGB0/4Kr9UDBnWDPsH92E6lRa5QlzDOolEqgFHyyRP1JYJH3RGKVlYK
rcLlluADiRYXCadwtXvnkJGxfg2DGICn5bEInPtM+bEhO3IfqrjipvT/Qx3/N6T+
hiHyl2Yyi0loUhbWsTuuSz+D07wj/4X1evuaaAc56RSwv0x6rLSjkYj1I7V3nMvc
e2fwNFiJvLdGfMcIYyxrOwO24cFwzYMYoTDFmf8MkN/H/khKZiksdnIxfcBFfyWu
PA8s5O3Zs90Ack3IvK7uAhRDz1PpR6Y+1bkCDQRjvZKEARAAuTgK6INJWBEzfrDM
vM157ZGAM/7pyevj0WCDhqiCFdpH3MVt7+wq0tmR8Oo5Lt4AXqVtzn1bw1sMAkWK
U6yxLtS7cMiXOAPOtemTzWQkvk9o1FFygRQ8oyp4RUP4wj+W4lYaDhY+tJRDr/sR
6grYt/lZbfvEPuxL4jGW/dLSKHTLs8kh367Xm1qxqaG1C1tSLusTPb/8uNpOCANh
A2HAJRCGMox7f295+mEWXujif8yIfYtSQldqh+2bA6vaV3WKtHTPdLa1zzB20rf9
Mguz4ff3XDJCHPWOKeBOfqVS9CL67TZeOx0nJ6u2JnNDlwlzX7R63v1D/tSTYzPL
mJeosIjpRQg4ELyyLSkj0lANvY/AwlKeTPkvoc76UwsQRFgxx6ZZjKObjAok6TQK
HjszRNkeBWbbi8J+zvfS6U3+1qYtvf9Enpp1v1CWfEKZmC68MgspNCzLSOpkoAfe
k2iQ/XsjKXSsaUXY5A1DljQTVbSs9G3OkQA0Eyv4JPj2KEXPoF/0sIt2QRrayyqk
1lqN4k9a3zEZ2WpkQLIRK5DgCE/ORHXkperEWrDiAfSvuVl999jxr+Jqi8qvlPrm
aQd0X5Wc5gpb7X72FMsb2UHaWsUEs6nwoAWnXgA3PGd0r9LihZMJXfMc+LSF/dRK
fx+PizkTXQbfML8fi7Il9JA1p4UAEQEAAYkEcgQYAQgAJhYhBPJtigqt9mbMso8q
sWUOwyU7ao/1BQJjvZKEAhsCBQkDwmcAAkAJEGUOwyU7ao/1wXQgBBkBCAAdFiEE
ytw9HjtcTF+Upl14p79lqq27xJQFAmO9koQACgkQp79lqq27xJQG2hAAp4813NAu
AOg4C/Yvq8aqnDRDHw/ISs5XsQTfVwbIssSiSTqdJb4jX0rbKW1qzM6l15EmEsPV
5MCGfN8xfP5+UeeVIJaXLq3BMYJf1An8sun9f8Bp2Wdw6IDlr9VwFZ170JQ2xYvq
VJ+s/rxbCJ8K9neDPelzN/KXMyUV/uA5D1G92IlItinw4ZqD9e/CjPfIBwfNEMnZ
nYaku4VGJfzaMHezaUTB8UVyFVN6Zv2PGYEUBCwISM61IdnGKnJza0NMnEvGstXN
vtnWk7H/12Q4/rDpApy68Qbuo8gbZIifjNY00u2iyx4BEvji418NfTdF5HuPHR4m
g10cz+FcWxo13PGTXHKprNC9Y4M5nMAZW8z05/2geD8jzmY9Yz3m0GSVF/0cD3pB
rQ/LXirxgJ2prCuE7Ax8XTTBg7+cjgqk0InKh2pF0sK+2UCbnN4hR+SQvR256hWI
F+TP/rDryaqdubqCOh7kytPnPqZtL8VqK7yDRhfmgxv3+bpvm+B2qm1okUCkH3bb
AkvowTBOcyTqLw7hYsREHkYVROYg57GGhMStkzaD+lep9kEUgcaXZF41W02WJeS3
VYXwooxFBKMhzm+cluLV+ujC+FnRslh7q/u90+3N2VljEjxA4Oj3RNAARzpOs0V2
BtuUsiPCTvhRLBmdG3RH25jm2hUPexP2+pMyEw//V211M6+MT5a8kCybK5e93I3+
eT2bfAfd1k0kcQcfbocymxW5DJUqHgBj+G9ZC5PIAeFk+Jfld0y3M186NAvP8I4+
ZNsJExdQyp1CN53mSWtxAadgHNNhDKX0KwyCarCk04xbf0qjlsrWNbsUI04sM1zt
C46N/0JsCuG4uAztAfU9hjbLmSxpjf04Qqpc5NDlGLgZ2xQTVmXPlFg1DgrF6fIq
WZwPa7z1eihkrEERPjnisjuwMd4uO5BIkqh8F7HdOnARYXpftg9LReV973z7i8n9
4rhpBedAHwVRqWo8owM8DOVTaHAQzMnnzB+6nCoOcZc7PzhWtKKhZupW2DYaLdIh
nlVCrmMSozkFn3shtOJ76XF2DMDpk0353w6i6rKghWC7TdpXPnWkHkExw4Pjnlse
1NP2vdz183NKqEKros463i+hOszQj7jb5DiFxxOnKUfxBNEMJXTqYzXdEzw7Sncw
NwTv4pFxnk3XFJD3IIXMdaCDYmHIJYK5Fwgc0Cop3dRAMJIB+0Q1/p+urDXqZphq
AGroZ22Z1DXzv7rm1x2drZyOBohc+dqn3zjEx+lwZ6CY8XPiQgbWEzSzY8YT4oUA
xRcs9cJ+0SK/HhW/EG51YNbr5IMDb3HvycHEreszEvwq2HdnsMIYdM8GC7fl7Zpp
0r+S1089BYMqKmhepps=
=srz3
-----END PGP PUBLIC KEY BLOCK-----
EOF

)
```

1.6 Getting Started with Daml

The goal of this tutorial is to get you up and running with full-stack Daml development. Through the example of a simple social networking application, you will learn:

1. How to build and run the application
2. The design of its different components (*App Architecture*)
3. How to write a new feature for the app (*Your First Feature*)

The goal is that by the end of this tutorial, you'll have a good idea of the following:

> What Daml contracts and ledgers are
> How a user interface (UI) interacts with a Daml ledger
> How Daml helps you build a real-life application fast.

This is not a comprehensive guide to all Daml concepts and tools or all deployment options; these are covered in-depth in the User Guide. **For a quick overview of the most important Daml concepts used in this tutorial you can refer to** the Daml cheat-sheet .

With that, let's get started!

1.6.1 Prerequisites

Make sure that you have the Daml SDK, Java 11 or higher, and Visual Studio Code (the only supported IDE) installed as per the instructions in *Installing the SDK*.

You will also need some common software tools to build and interact with the template project:

> Node and the associated package manager `npm`. Use the Active LTS Node version, currently `v18` (check with `node --version`).
> A terminal application for command line interaction.

1.6.2 Run the App

To get the app up and running:

1. Open a terminal, select a folder in which to create your first application, and instantiate the template project.

```
daml new create-daml-app --template create-daml-app
```

This creates a new folder with contents from our template. To see a list of all available templates run `daml new --list`.

2. Change to the new folder:

```
cd create-daml-app
```

3. Open two terminal windows.
4. In one terminal, at the root of the `create-daml-app` directory, run the command:

```
daml start
```

Any commands starting with `daml` are using the *Daml Assistant*, a command line tool in the SDK for building and running Daml apps.

The command has started successfully when you see the `INFO com.daml.http.Main$ - Started server: ServerBinding(/127.0.0.1:7575)` message in the terminal. The command does a few things:

1. Compiles the Daml code to a DAR (Daml Archive) file
2. Generates a JavaScript library in `ui/daml.js` to connect the UI with your Daml code
3. Starts an instance of the *Sandbox*, an in-memory ledger useful for development, loaded with our DAR
4. Starts a server for the *HTTP JSON API*, a simple way to run commands against a Daml ledger (in this case the running Sandbox)

We'll leave these processes running to serve requests from our UI.

5. In the second terminal, navigate to the `create-daml-app/ui` folder and use `npm` to install the project dependencies:

```
cd create-daml-app/ui
npm install
```

This step may take a couple of moments. You should see `success Saved lockfile.` in the output if everything worked as expected.

6. Start the UI with:

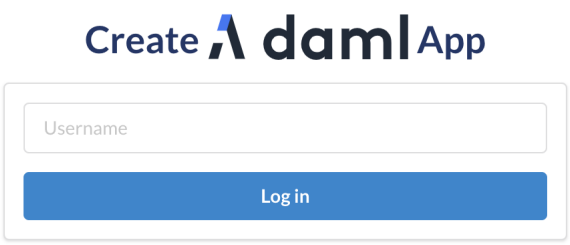
```
npm start
```

This starts the web UI connected to the running Sandbox and JSON API server. The command should automatically open a window in your default browser at http://localhost:3000.

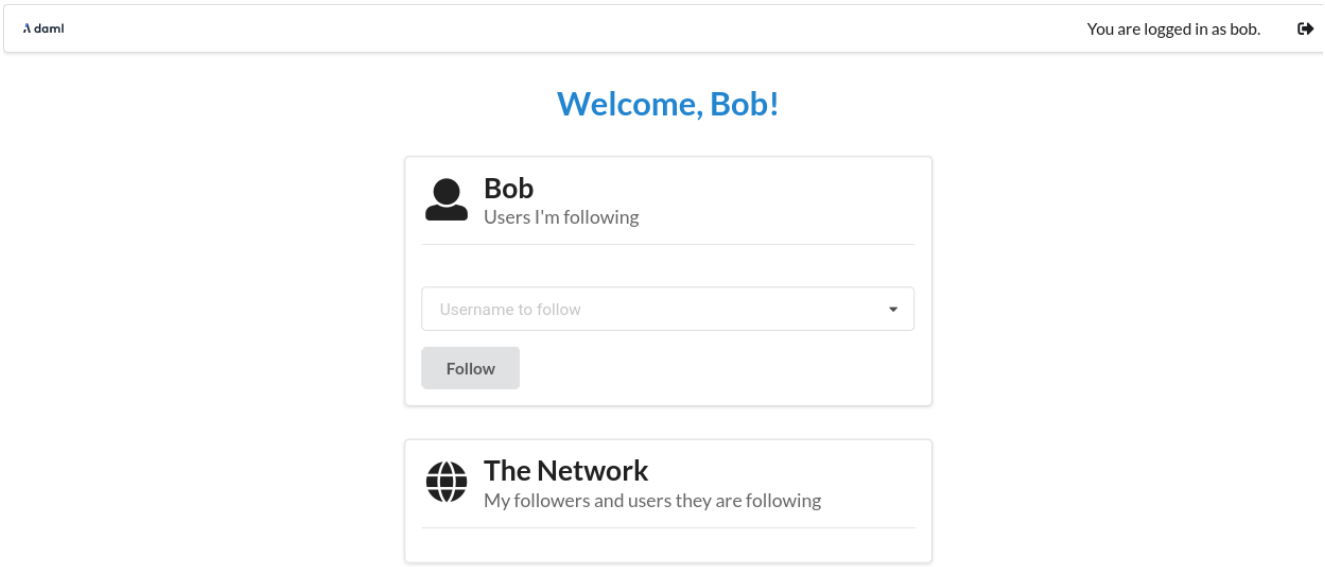
Once the web UI has been compiled and started, you should see `Compiled successfully!` in your terminal. If you don't, open http://localhost:3000 in a web browser. Depending on your firewall settings, you may be asked whether to allow the app to receive network connections. It is safe to accept.

You should now see the login page for the social network. For simplicity, in this app there is no password or sign-up required.

1. Enter a user name. Valid user names are bob, alice, or charlie (note that these are all lower-case, although they are displayed in the social network UI by their alias instead of their user id, with the usual capitalization).
2. Click *Log in*.



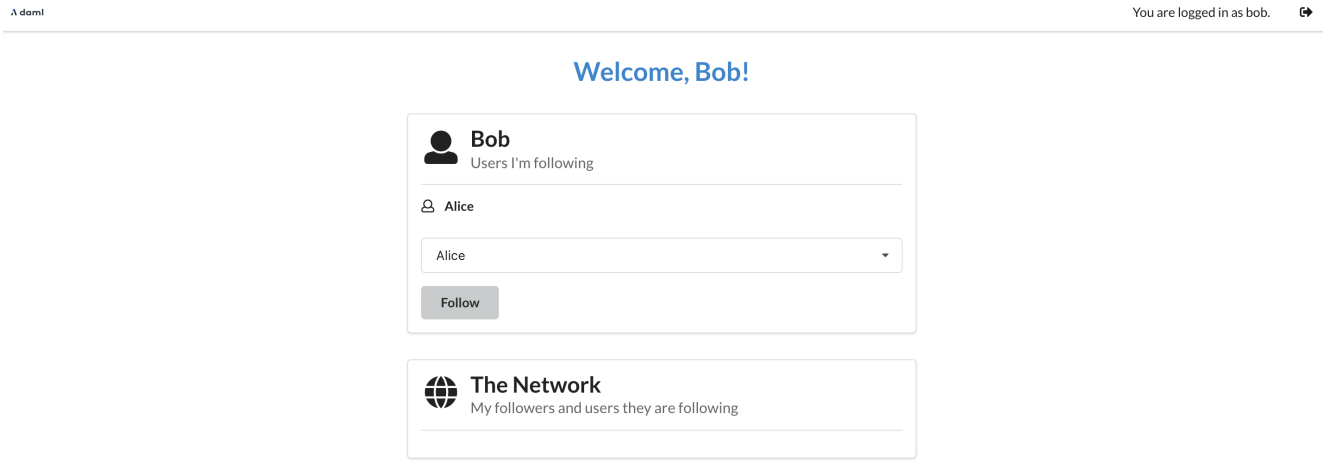
You should see the main screen with two panels. The top panel displays the social network users you are following; the bottom displays the aliases of the users who follow you. Initially these are both empty as you are not following anyone and you don't have any followers. To start following a user, select their name in the drop-down list and click the *Follow* button in the top panel. At the moment, you will notice that the drop-down shows only your own user because no other user has registered yet.



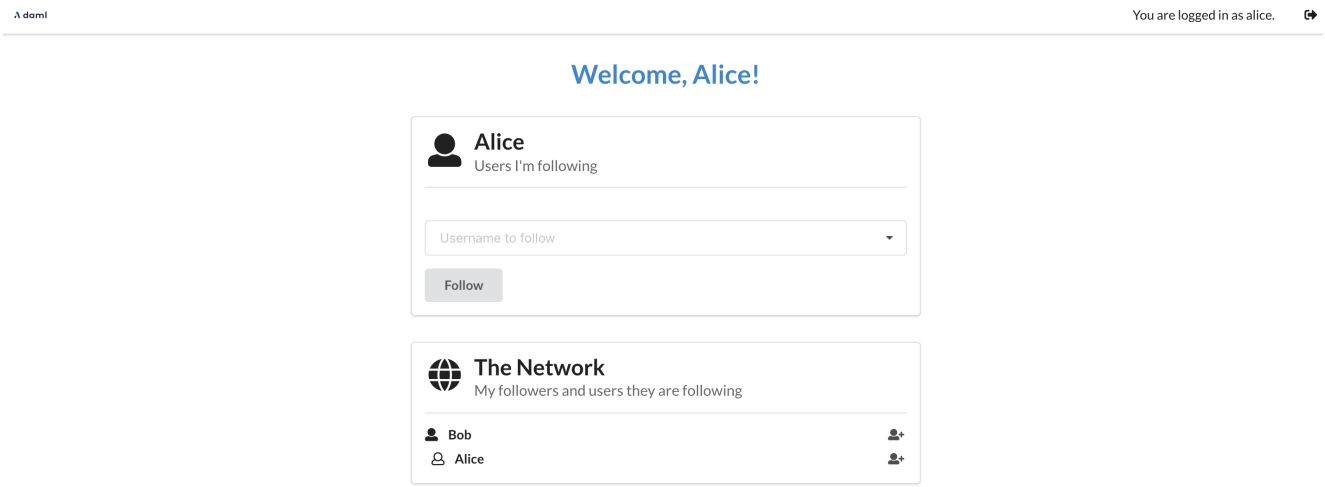
Next, open a new browser window/tab at http://localhost:3000 and log in as a different user. (Having separate windows/tabs allows you to see both your own screen and the screen of the user you are following at the same time.)

Now that the other user (Alice in this example) has logged in, go back to the previous window/tab, select them drop-down list and click the *Follow* button in the top panel.

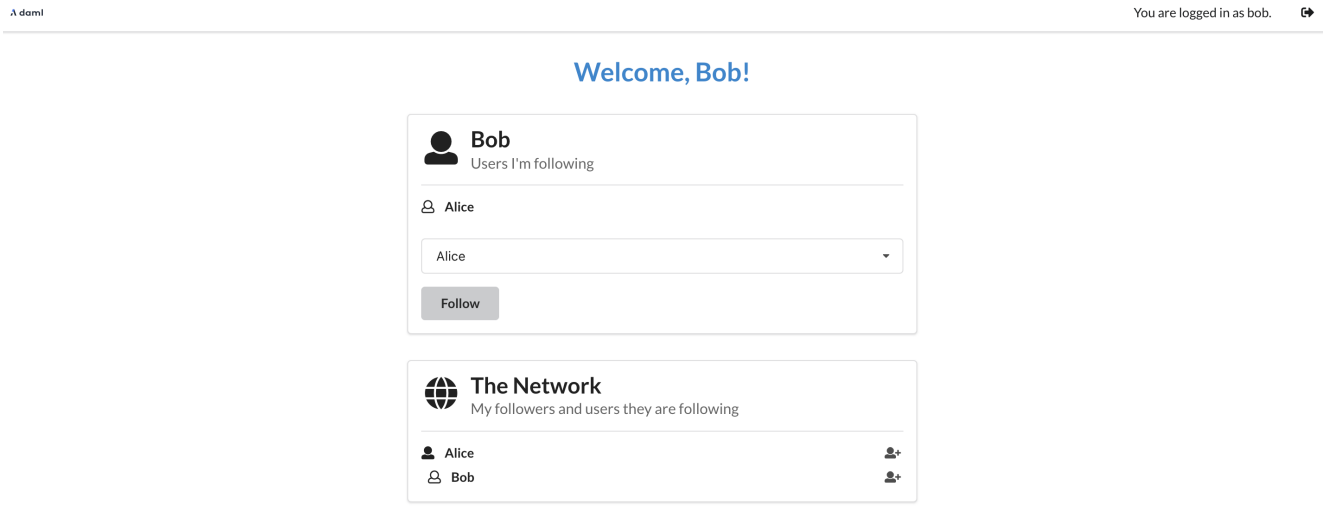
The user you just started following appears in the *Following* panel. However, they do not yet appear in the *Network* panel. This is because they have not yet started following you. This social network is similar to Twitter and Instagram, where by following someone, say Alice, you make yourself visible to her but not vice versa. We will see how we encode this in Daml in the next section.



To make this relationship reciprocal, go back to the other window/tab where you logged in as the second user (Alice in this example). You should now see your name in her network. In fact, Alice can see the entire list of users you are following in the *Network* panel. This is because this list is part of the user data that became visible when you started following her.



When Alice starts following you, you can see her in your network as well. Switch to the window where you are logged in as yourself - the network should update automatically.



Play around more with the app at your leisure: create new users and start following more users. Observe when a user becomes visible to others - this will be important to understanding Daml's privacy model later. When you're ready, let's move on to the *architecture of our app*.

Tip: Congratulations on completing the first part of the Getting Started Guide! Join our forum and share a screenshot of your accomplishment to get your first of 3 getting started badges! You can get the next one by *implementing your first feature*.

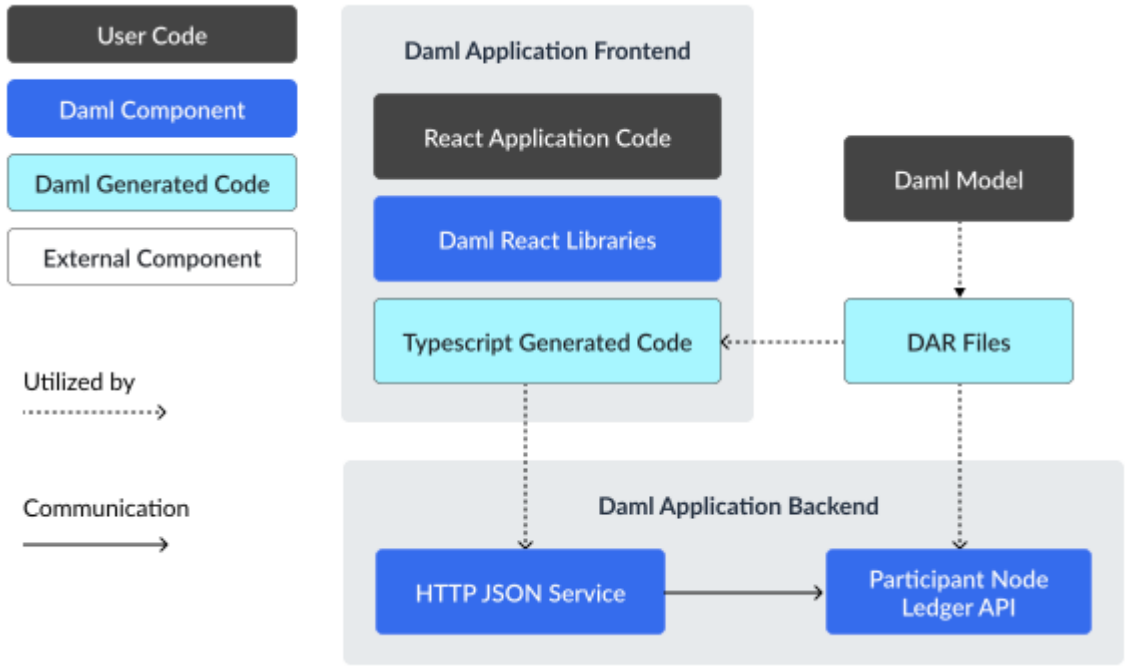
1.7 App Architecture

In this section we'll look at the different components of the social network app we created in Building Your App. The goal is to familiarize yourself with the basics of Daml architecture enough to feel comfortable extending the code with a new feature in the next section. There are two main components:

the Daml model
the React/TypeScript frontend

We generate TypeScript code to bridge the two.

Overall, the social networking app follows the *recommended architecture of a fullstack Daml application*. Below is a simplified version of the architecture represented in the app.



There are four types of building blocks that go into our application: user code, generated code from Daml, Daml components, and external components. The Daml model determines the DAR files that underpin both the frontend and backend. The frontend includes React application code, Daml React libraries, and Typescript generated code. From the client point of view, the backend consists of the JSON API and a participant node.

Let's start by looking at the Daml model, which defines the core logic of the application. Have the Daml cheat-sheet open in a separate tab for a quick overview of the most common Daml concepts.

1.7.1 The Daml Model

In your terminal, navigate to the root `create-daml-app` directory and run:

```
daml studio
```

This should open the Visual Studio Code editor at the root of the project. (You may get a new tab pop up with release notes for the latest version of Daml - close this.) Using the file *Explorer* on the left sidebar, navigate to the `daml` folder and double-click on the `User.daml` file.

The Daml code defines the *data* and *workflow* of the application. Both are described in the `User` contract *template*. Let's look at the data portion first:

```
template User with
    username: Party
    following: [Party]
  where
    signatory username
    observer following
```

There are two important aspects here:

1. The data definition (a *schema* in database terms), describing the data stored with each user contract. In this case it is an identifier for the user and the list of users they are following. Both fields use the built-in `Party` type which lets us use them in the following clauses.

2. The signatories and observers of the contract. The signatories are the parties whose authorization is required to create or archive contracts, in this case the user herself. The observers are the parties who are able to view the contract on the ledger. In this case all users that a particular user is following are able to see the user contract.

It's also important to distinguish between parties, users, and aliases in terms of naming:

Parties are unique across the entire Daml network. These must be allocated before you can use them to log in, and allocation results in a random-looking (but not actually random) string that identifies the party and is used in your Daml code. Parties are a builtin concept.
On each participant node you can create users with human-readable user ids. Each user can be associated with one or more parties allocated on that participant node, and refers to that party only on that node. Users are a purely local concept, meaning you can never address a user on another node by user id, and you never work with users in your Daml code; party ids are always used for these purposes. Users are also a builtin concept.
Lastly we have user aliases. These are not a builtin concept, they are defined by an *Alias template* (discussed below) within the specific model used in this guide. Aliases serve as a way to address parties on all nodes via a human readable name.

The social network users discussed in this guide are really a combination of all three of these concepts. Alice, Bob, and Charlie are all aliases that correspond to a single test user and a single party id each. As part of running *daml start*, the *init-script* specified in *daml.yaml* is executed. This points at the *Setup:setup* function which defines a *Daml Script* which creates 3 users *alice*, *bob* and *charlie* as well as a corresponding party for each they can act as. In addition to that, we also create a separate public party and allow the three users to read contracts for that party. This allows us to share the alias contracts with that public party and have them be visible to all 3 users.

Now let's see what the `signatory` and `observer` clauses mean in our app in more concrete terms. The user with the alias Alice can see another user, alias Bob, in the network only when Bob is following Alice (only if Alice is in the `following` list in his user contract). For this to be true, Bob must have

previously started to follow Alice, as he is the sole signatory on his user contract. If not, Bob will be invisible to Alice.

This illustrates two concepts that are central to Daml: *authorization* and *privacy*. Authorization is about who can *do* what, and privacy is about who can *see* what. In Daml you must answer these questions upfront, as they are fundamental to the design of the application.

The next part of the Daml model is the operation to follow users, called a *choice* in Daml:

```
    nonconsuming choice Follow: ContractId User with
        userToFollow: Party
      controller username
      do
        assertMsg "You cannot follow yourself" (userToFollow /= username)
        assertMsg "You cannot follow the same user twice" (notElem userToFollow
↪following)
        archive self
        create this with following = userToFollow :: following
```

Daml contracts are *immutable* (can not be changed in place), so the only way to update one is to archive it and create a new instance. That is what the `Follow` choice does: after checking some preconditions, it archives the current user contract and creates a new one with the new user to follow added to the list. Here is a quick explanation of the code:

> The choice starts with the `nonconsuming choice` keyword followed by the choice name `Follow`.
> The return type of a choice is defined next. In this case it is `ContractId User`.
> After that we declare choice parameters with the `with` keyword. Here this is the user we want to start following.
> The keyword `controller` defines the `Party` that is allowed to execute the choice. In this case, it is the `username` party associated with the `User` contract.
> The `do` keyword marks the start of the choice body where its functionality will be written.
> After passing some checks, the current contract is archived with `archive self`.
> A new `User` contract with the new user we have started following is created (the new user is added to the `following` list).

More detailed information on choices can be found in *our docs*.

Finally, the `User.daml` file contains the Alias template that manages the link between user ids and their aliases. The alias template sets the public party we created in the setup script as the observer of the contract. Because we allow all users to read contracts visible to the public party, this allows e.g., Alice to see Bob's *Alias* contract.

```
template Alias with
    username: Party
    alias: Text
    public: Party
  where
    signatory username
    observer public

    key (username, public) : (Party, Party)
    maintainer key._1

    nonconsuming choice Change: ContractId Alias with
        newAlias: Text
```

(continues on next page)

(continued from previous page)

```
      controller username
      do
        archive self
        create this with alias = newAlias
```

Let's move on to how our Daml model is reflected and used on the UI side.

1.7.2 TypeScript Code Generation

The user interface for our app is written in TypeScript. TypeScript is a variant of JavaScript that provides more support during development through its type system.

To build an application on top of Daml, we need a way to refer to our Daml templates and choices in TypeScript. We do this using a Daml to TypeScript code generation tool in the SDK.

To run code generation, we first need to compile the Daml model to an archive format (a `.dar` file). The `daml codegen js` command then takes this file as argument to produce a number of TypeScript packages in the output folder.

```
daml build
daml codegen js .daml/dist/create-daml-app-0.1.0.dar -o daml.js
```

Now we have a TypeScript interface (types and companion objects) to our Daml model, which we'll use in our UI code next.

1.7.3 The UI

On top of TypeScript, we use the UI framework React. React helps us write modular UI components using a functional style - a component is rerendered whenever one of its inputs changes - with careful use of global state.

Let's see an example of a React component. All components are in the `ui/src/components` folder. You can navigate there within Visual Studio Code using the file explorer on the left sidebar. We'll first look at `App.tsx`, which is the entry point to our application.

```
const App: React.FC = () => {
  const [credentials, setCredentials] = React.useState<
    Credentials | undefined
  >();
  if (credentials) {
    const PublicPartyLedger: React.FC = ({ children }) => {
      const publicToken = usePublicToken();
      const publicParty = usePublicParty();
      if (publicToken && publicParty) {
        return (
          <publicContext.DamlLedger
            token={publicToken.token}
            party={publicParty}>
            {children}
          </publicContext.DamlLedger>
        );
      } else {
```

(continues on next page)

(continued from previous page)

```
        return <h1>Loading ...</h1>;
      }
    };
    const Wrap: React.FC = ({ children }) =>
      isRunningOnHub() ? (
        <DamlHub token={credentials.token}>
          <PublicPartyLedger>{children}</PublicPartyLedger>
        </DamlHub>
      ) : (
        <div>{children}</div>
      );
    return (
      <Wrap>
        <userContext.DamlLedger
          token={credentials.token}
          party={credentials.party}
          user={credentials.user}>
          <MainScreen
            getPublicParty={credentials.getPublicParty}
            onLogout={() => {
              if (authConfig.provider === "daml-hub") {
                damlHubLogout();
              }
              setCredentials(undefined);
            }}
          />
        </userContext.DamlLedger>
      </Wrap>
    );
  } else {
    return <LoginScreen onLogin={setCredentials} />;
  }
};
```

An important tool in the design of our components is a React feature called Hooks. Hooks allow you to share and update state across components, avoiding the need to thread it through manually. We take advantage of hooks to share ledger state across components. Custom *Daml React hooks* query the ledger for contracts, create new contracts, and exercise choices. This is the library you will use most often when interacting with the ledger[1] .

The `useState` hook (not specific to Daml) here keeps track of the user's credentials. If they are not set, we render the `LoginScreen` with a callback to `setCredentials`. If they are set, we render the `MainScreen` of the app. This is wrapped in the `DamlLedger` component, a React context with a handle to the ledger.

Let's move on to more advanced uses of our Daml React library. The `MainScreen` is a simple frame around the `MainView` component, which houses the main functionality of our app. It uses Daml React hooks to query and update ledger state.

```
const MainView: React.FC = () => {
  const username = userContext.useParty();
  const myUserResult = userContext.useStreamFetchByKeys(User.User, () =>
↪[username], [username]);
```

(continues on next page)

[1] Behind the scenes the Daml React hooks library uses the *Daml Ledger TypeScript library* to communicate with a ledger implementation via the *HTTP JSON API*.

(continued from previous page)

```
const aliases = publicContext.useStreamQueries(User.Alias, () => [], []);
const myUser = myUserResult.contracts[0]?.payload;
const allUsers = userContext.useStreamQueries(User.User).contracts;
```

The `useParty` hook returns the current user as stored in the `DamlLedger` context. A more interesting example is the `allUsers` line. This uses the `useStreamQueries` hook to get all `User` contracts on the ledger. (`User.User` here is an object generated by `daml codegen js` - it stores metadata of the `User` template defined in `User.daml`.) Note however that this query preserves privacy: only users that follow the current user have their contracts revealed. This behaviour is due to the observers on the `User` contract being exactly in the list of users that the current user is following.

A final point on this is the *streaming* aspect of the query. Results are updated as they come in - there is no need for periodic or manual reloading to see updates.

Another example, showing how to *update* ledger state, is how we exercise the `Follow` choice of the `User` template.

```
const ledger = userContext.useLedger();

const follow = async (userToFollow: Party): Promise<boolean> => {
  try {
    await ledger.exerciseByKey(User.User.Follow, username, {userToFollow});
    return true;
  } catch (error) {
    alert(`Unknown error:\n${JSON.stringify(error)}`);
    return false;
  }
}
```

The `useLedger` hook returns an object with methods for exercising choices. The core of the `follow` function here is the call to `ledger.exerciseByKey`. The *key* in this case is the username of the current user, used to look up the corresponding `User` contract. The wrapper function `follow` is then passed to the subcomponents of `MainView`. For example, `follow` is passed to the `UserList` component as an argument (a prop in React terms). This is triggered when you click the icon next to a user's name in the *Network* panel.

```
            <UserList
              users={followers}
              partyToAlias={partyToAlias}
              onFollow={follow}
            />
```

This should give you a taste of how the UI works alongside a Daml ledger. You'll see this more as you develop *your first feature* for our social network.

1.8 Your First Feature

To get a better idea of how to develop Daml applications, let's try implementing a new feature for our social network app.

At the moment, our app lets us follow users in the network, but we have no way to communicate with them. Let's fix that by adding a *direct messaging* feature. This should let users that follow each other send messages to each other, respecting *authorization* and *privacy*. This means:

> You cannot send a message to someone unless they have given you the authority by following you back.
> You cannot see a message unless you sent it or it was sent to you.

Daml lets us implement these guarantees in a direct and intuitive way.

Creating a feature involves three steps:

1. Adding the necessary changes to the Daml model
2. Making the corresponding changes in the UI
3. Running the app with the new feature

As usual, we must start with the Daml model and base our UI changes on top of that.

1.8.1 Daml Changes

The Daml code defines the *data* and *workflow* of the application; you can read about this in more detail in the *architecture* section. The workflow refers to the interactions between parties that are permitted by the system. In the context of a messaging feature, these are essentially the authorization and privacy concerns listed above.

For the authorization part, we take the following approach: a user Bob can message another user Alice when Alice starts following Bob back. When Alice starts following Bob back, she gives permission or *authority* to Bob to send her a message.

To implement this workflow, let's start by adding the new *data* for messages. Navigate to the `daml/User.daml` file and copy the following `Message` template to the bottom. Indentation is important: it should be at the top level like the original `User` template.

```
template Message with
    sender: Party
    receiver: Party
    content: Text
  where
    signatory sender, receiver
```

This template is very simple: it contains the data for a message and no choices. The interesting part is the `signatory` clause: both the `sender` and `receiver` are signatories on the template. This enforces that creation and archival of `Message` contracts must be authorized by both parties.

Now we can add messaging into the workflow by adding a new choice to the `User` template. Copy the following choice to the `User` template after the `Follow` choice. The indentation for the `SendMessage` choice must match the one of `Follow`. *Make sure you save the file after copying the code.*

```
    nonconsuming choice SendMessage: ContractId Message with
        sender: Party
```

(continues on next page)

(continued from previous page)

```
        content: Text
      controller sender
      do
        assertMsg "Designated user must follow you back to send a message" (elem
↪sender following)
        create Message with sender, receiver = username, content
```

As with the `Follow` choice, there are a few aspects to note here.

By convention, the choice returns the `ContractId` of the resulting `Message` contract.
The parameters to the choice are the `sender` and `content` of this message; the receiver is the party named on this `User` contract.
The `controller` clause states that it is the `sender` who can exercise the choice.
The body of the choice first ensures that the sender is a user that the receiver is following and then creates the `Message` contract with the `receiver` being the signatory of the `User` contract.

This completes the workflow for messaging in our app.

Navigate to the terminal window where the `daml start` process is running and press 'r'. This will

Compile our Daml code into a *DAR file containing the new feature*
Update the JavaScript library under `ui/daml.js` to connect the UI with your Daml code
Upload the *new DAR file* to the sandbox

As mentioned previously, Daml Sandbox uses an in-memory store, which means it loses its state – which here includes all user data and follower relationships – when stopped or restarted.

Now let's integrate the new functionality into the UI.

1.8.2 Messaging UI

The UI for messaging consists of a new *Messages* panel in addition to the *Follow* and *Network* panel. This new panel has two parts:

1. A list of messages you've received with their senders.
2. A form with a dropdown menu for follower selection and a text field for composing the message.

We implement each part as a React component, named `MessageList` and `MessageEdit` respectively. Let's start with the simpler `MessageList`.

1.8.2.1 MessageList Component

The goal of the `MessageList` component is to query all `Message` contracts where the `receiver` is the current user, and display their contents and senders in a list. The entire component is shown below. Copy this into a new `MessageList.tsx` file in `ui/src/components` and save it.

```
import React from 'react'
import { List, ListItem } from 'semantic-ui-react';
import { User } from '@daml.js/create-daml-app';
import { userContext } from './App';

type Props = {
  partyToAlias: Map<string, string>
```

(continues on next page)

(continued from previous page)

```
}
/**
 * React component displaying the list of messages for the current user.
 */
const MessageList: React.FC<Props> = ({partyToAlias}) => {
  const messagesResult = userContext.useStreamQueries(User.Message);

  return (
    <List relaxed>
      {messagesResult.contracts.map(message => {
        const {sender, receiver, content} = message.payload;
        return (
          <ListItem
            className='test-select-message-item'
            key={message.contractId}>
            <strong>{partyToAlias.get(sender) ?? sender} &rarr; {partyToAlias.
↪get(receiver) ?? receiver}:</strong> {content}
          </ListItem>
        );
      })}
    </List>
  );
};

export default MessageList;
```

In the component body, `messagesResult` gets the stream of all `Message` contracts visible to the current user. The streaming aspect means that we don't need to reload the page when new messages come in. For each contract in the stream, we destructure the *payload* (the data as opposed to metadata like the contract ID) into the `{sender, receiver, content}` object pattern. Then we construct a `ListItem` UI element with the details of the message.

An important point about privacy: no matter how we write our `Message` query in the UI code, it is impossible to break the privacy rules given by the Daml model. That is, it is impossible to see a `Message` contract of which you are not the `sender` or the `receiver` (the only parties that can observe the contract). This is a major benefit of writing apps on Daml: the burden of ensuring privacy and authorization is confined to the Daml model.

1.8.2.2 MessageEdit Component

Next we need the `MessageEdit` component to compose and send messages to our followers. Again we show the entire component here; copy this into a new `MessageEdit.tsx` file in `ui/src/components` and save it.

```
import React from 'react'
import { Form, Button } from 'semantic-ui-react';
import { Party } from '@daml/types';
import { User } from '@daml.js/create-daml-app';
import { userContext } from './App';

type Props = {
  followers: Party[];
  partyToAlias: Map<string, string>;
```

(continues on next page)

(continued from previous page)

```
}

/**
 * React component to edit a message to send to a follower.
 */
const MessageEdit: React.FC<Props> = ({followers, partyToAlias}) => {
  const sender = userContext.useParty();
  const [receiver, setReceiver] = React.useState<string | undefined>();
  const [content, setContent] = React.useState("");
  const [isSubmitting, setIsSubmitting] = React.useState(false);
  const ledger = userContext.useLedger();

  const submitMessage = async (event: React.FormEvent) => {
    try {
      event.preventDefault();
      if (receiver === undefined) {
        return;
      }
      setIsSubmitting(true);
      await ledger.exerciseByKey(User.User.SendMessage, receiver, {sender,
↪content});
      setContent("");
    } catch (error) {
      alert(`Error sending message:\n${JSON.stringify(error)}`);
    } finally {
      setIsSubmitting(false);
    }
  };

  return (
    <Form onSubmit={submitMessage}>
      <Form.Select
        fluid
        search
        className='test-select-message-receiver'
        placeholder={receiver ? partyToAlias.get(receiver) ?? receiver : "Select
↪a follower"}
        value={receiver}
        options={followers.map(follower => ({ key: follower, text: partyToAlias.
↪get(follower) ?? follower, value: follower }))}
        onChange={(event, data) => setReceiver(data.value?.toString())}
      />
      <Form.Input
        className='test-select-message-content'
        placeholder="Write a message"
        value={content}
        onChange={event => setContent(event.currentTarget.value)}
      />
      <Button
        fluid
        className='test-select-message-send-button'
        type="submit"
        disabled={isSubmitting || receiver === undefined || content === ""}
        loading={isSubmitting}
        content="Send"
      />
```

(continues on next page)

(continued from previous page)

```
    </Form>
  );
};

export default MessageEdit;
```

You will first notice a `Props` type near the top of the file with a single `followers` field. A *prop* in React is an input to a component; in this case a list of users from which to select the message receiver. The prop will be passed down from the `MainView` component, reusing the work required to query users from the ledger. You can see this `followers` field bound at the start of the `MessageEdit` component.

We use the React `useState` hook to get and set the current choices of message `receiver` and `content`. The Daml-specific `useLedger` hook gives us an object we can use to perform ledger operations. The call to `ledger.exerciseByKey` in `submitMessage` looks up the `User` contract with the receiver's username and exercises the `SendMessage` choice with the appropriate arguments. If the choice fails, the `catch` block reports the error in a dialog box. Additionally, `submitMessage` sets the `isSubmitting` state so that the *Send* button is disabled while the request is processed. The result of a successful call to `submitMessage` is a new `Message` contract created on the ledger.

The return value of this component is the React `Form` element. This contains a dropdown menu to select a receiver from the `followers`, a text field for the message content, and a *Send* button which triggers `submitMessage`.

Note how *authorization* is enforced here. Due to the logic of the `SendMessage` choice, it is impossible to send a message to a user who is not following us (even if you could somehow access their `User` contract). The assertion that `elem sender following` in `SendMessage` ensures this: no mistake or malice by the UI programmer could breach this.

1.8.2.3 MainView Component

Finally we can see these components come together in the `MainView` component. We want to add a new panel to house our messaging UI. Open the `ui/src/components/MainView.tsx` file and start by adding imports for the two new components.

```
import MessageEdit from './MessageEdit';
import MessageList from './MessageList';
```

Next, find where the *Network* `Segment` closes, towards the end of the component. This is where we'll add a new `Segment` for *Messages*. Make sure you save the file after copying over the code.

```
            <Segment>
              <Header as='h2'>
                <Icon name='pencil square' />
                <Header.Content>
                  Messages
                  <Header.Subheader>Send a message to a follower</Header.
↪Subheader>
                </Header.Content>
              </Header>
              <MessageEdit
                followers={followers.map(follower => follower.username)}
                partyToAlias={partyToAlias}
```

(continues on next page)

(continued from previous page)

```
            />
            <Divider />
            <MessageList partyToAlias={partyToAlias}/>
          </Segment>
```

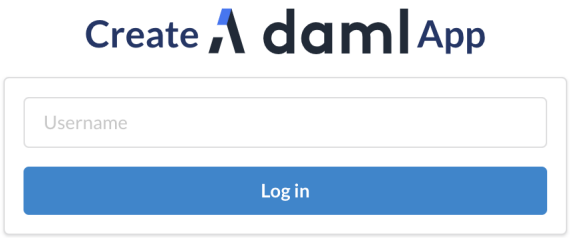
Following the formatting of the previous panels, we include the new messaging components: `MessageEdit` supplied with the usernames of all visible parties as props, and `MessageList` to display all messages.

That is all for the implementation! Let's give the new functionality a spin.

1.8.3 Run the Updated UI

If you have the frontend UI up and running you're all set. If you don't have the UI running, open a new terminal window and navigate to the `create-daml-app/ui` folder, then run the `npm start` command to start the UI.

You should see the same login page as before at http://localhost:3000.

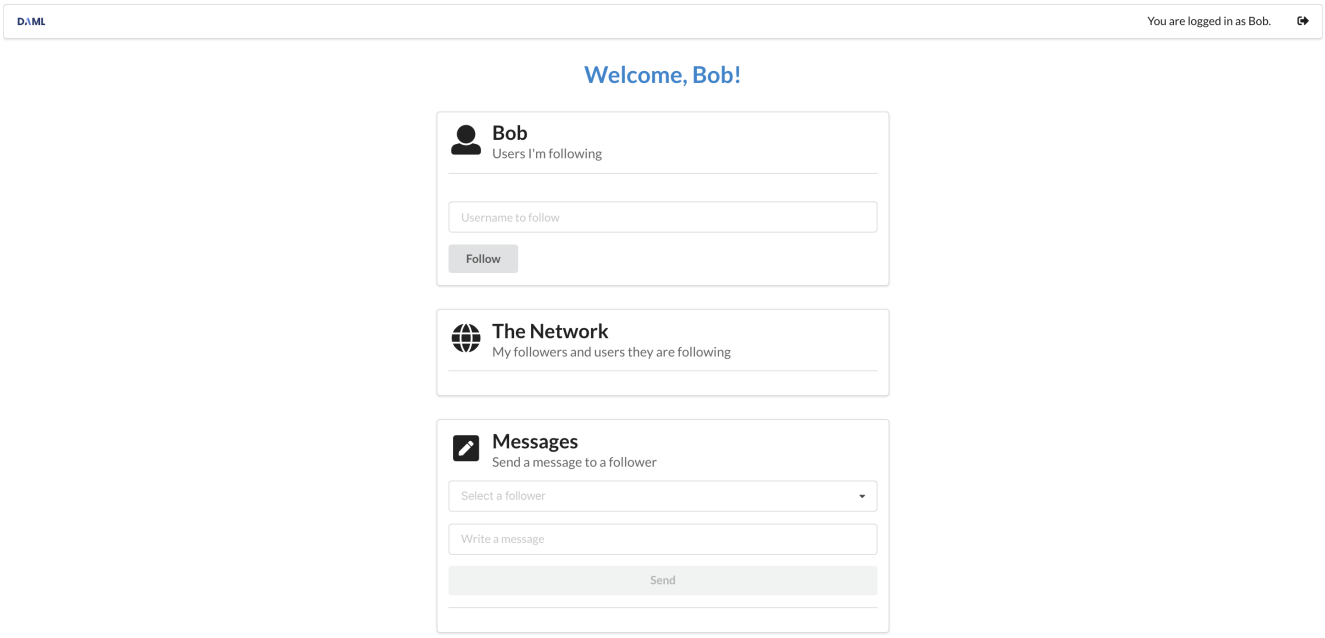


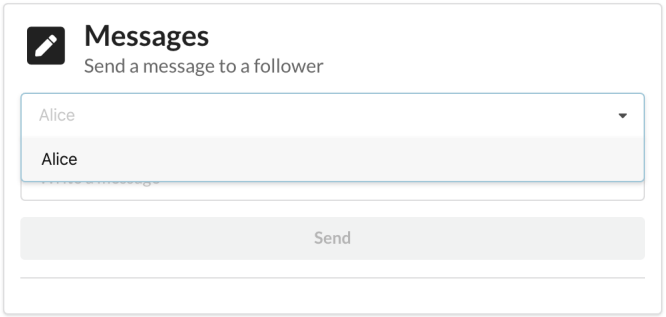
Once you've logged in, you'll see a familiar UI but with our new *Messages* panel at the bottom!

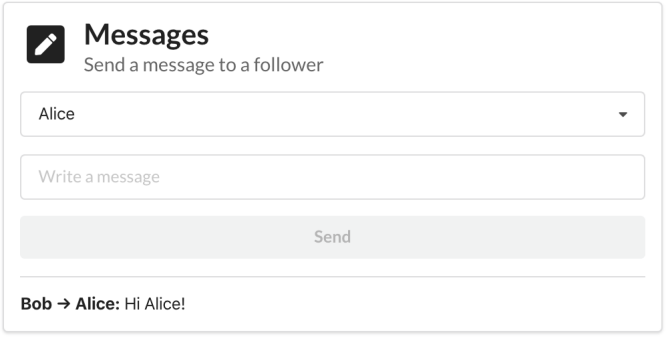
Go ahead and follow more users, and log in as some of those users in separate browser windows to follow yourself back. Then click on the dropdown menu in the *Messages* panel to see a choice of followers to message!

Send some messages between users and make sure you can see each one from the other side. Notice that each new message appears in the UI as soon as it is sent (due to the *streaming* React hooks).

Tip: You completed the second part of the Getting Started Guide! Join our forum and share a screenshot of your accomplishment to get your second of 3 badges! Get the third badge by deploying to Daml Hub.


DAML
You are logged in as Bob.
Welcome, Bob!
Bob
Users I'm following
Username to follow
Follow
The Network
My followers and users they are following
Messages
Send a message to a follower
Select a follower
Write a message
Send


Messages
Send a message to a follower
Alice
Alice
Send


Messages
Send a message to a follower
Alice
Write a message
Send
Bob → Alice: Hi Alice!

1.8.4 Next Steps

We've gone through the process of setting up a full-stack Daml app and implementing a useful feature end to end. As the next step we encourage you to really dig into the fundamentals of Daml and understand its core concepts such as parties, signatories, observers, and controllers. You can do that either by *going through our docs* or by taking an online course.

After you've got a good grip on these concepts learn *how to conduct end-to-end testing of your app*.

1.9 Testing Your Web App

When developing a UI for your Daml application, you will want to test that user flows work from end to end. This means that actions performed in the web UI trigger updates to the ledger and give the desired results on the page. In this section we show how you can do such testing automatically in TypeScript (equally JavaScript). This will allow you to iterate on your app faster and with more confidence!

There are two tools that we chose to write end to end tests for our app. Of course there are more to choose from, but this is one combination that works.

> Jest is a general-purpose testing framework for JavaScript that's well integrated with both TypeScript and React. Jest helps you structure your tests and express expectations of the app's behaviour.
> Puppeteer is a library for controlling a Chrome browser from JavaScript/TypeScript. Puppeteer allows you to simulate interactions with the app in place of a real user.

To install Puppeteer and some other testing utilities we are going to use, run the following command in the `ui` directory:

```
npm i --save-dev puppeteer@~10.0.0 wait-on@~6.0.1 @types/jest@~29.2.3 @types/
↪node@~18.11.9 @types/puppeteer@~7.0.4 @types/wait-on@~5.3.1
```

You may need to run `npm install` again afterwards.

Because these things are easier to describe with concrete examples, this section will show how to set up end-to-end tests for the application you would end with at the end of the *Your First Feature* section.

1.9.1 Set Up the Tests

Let's see how to use these tools to write some tests for our social network app. You can see the full suite in section *The Full Test Suite* at the bottom of this page. To run this test suite, create a new file `ui/src/index.test.ts`, copy the code in this section into that file and run the following command in the `ui` folder:

```
npm test
```

The actual tests are the clauses beginning with `test`. You can scroll down to the important ones with the following descriptions (the first argument to each `test`):

> 'log in as a new user, log out and log back in'
> 'log in as three different users and start following each other'
> 'error when following self'
> 'error when adding a user that you are already following'

Before this, we need to set up the environment in which the tests run. At the top of the file we have some global state that we use throughout. Specifically, we have child processes for the `daml start` and `npm start` commands, which run for the duration of our tests. We also have a single Puppeteer browser that we share among tests, opening new browser pages for each one.

The `beforeAll()` section is a function run once before any of the tests run. We use it to spawn the `daml start` and `npm start` processes and launch the browser. On the other hand the `afterAll()` section is used to shut down these processes and close the browser. This step is important to prevent child processes persisting in the background after our program has finished.

1.9.2 Example: Log In and Out

Now let's get to a test! The idea is to control the browser in the same way we would expect a user to in each scenario we want to test. This means we use Puppeteer to type text into input forms, click buttons and search for particular elements on the page. In order to find those elements, we do need to make some adjustments in our React components, which we'll show later. Let's start at a higher level with a `test`.

```
test("log in as a new user, log out and log back in", async () => {
  const [user, party] = await getParty();

  // Log in as a new user.
  const page = await newUiPage();
  await login(page, user);

  // Check that the ledger contains the new User contract.
  const token = insecure.makeToken(user);
  const ledger = new Ledger({ token });
  const users = await ledger.query(User.User);
  expect(users).toHaveLength(1);
  expect(users[0].payload.username).toEqual(party);

  // Log out and in again as the same user.
  await logout(page);
  await login(page, user);

  // Check we have the same one user.
  const usersFinal = await ledger.query(User.User);
  expect(usersFinal).toHaveLength(1);
  expect(usersFinal[0].payload.username).toEqual(party);

  await page.close();
}, 40_000);
```

We'll walk though this step by step.

> The `test` syntax is provided by Jest to indicate a new test running the function given as an argument (along with a description and time limit).
> `getParty()` gives us a new party name. Right now it is just a string unique to this set of tests, but in the future we will use the Party Management Service to allocate parties.
> `newUiPage()` is a helper function that uses the Puppeteer browser to open a new page (we use one page per party in these tests), navigate to the app URL and return a `Page` object.
> Next we `login()` using the new page and party name. This should take the user to the main screen. We'll show how the `login()` function does this shortly.

We use the `@daml/ledger` library to check the ledger state. In this case, we want to ensure there is a single `User` contract created for the new party. Hence we create a new connection to the `Ledger`, `query()` it and state what we `expect` of the result. When we run the tests, Jest will check these expectations and report any failures for us to fix.
The test also simulates the new user logging out and then logging back in. We again check the state of the ledger and see that it's the same as before.
Finally we must `close()` the browser page, which was opened in `newUiPage()`, to avoid runaway Puppeteer processes after the tests finish.

You will likely use `test`, `getParty()`, `newUiPage()` and `Browser.close()` for all your tests. In this case we use the `@daml/ledger` library to inspect the state of the ledger, but usually we just check the contents of the web page match our expectations.

1.9.3 Accessing UI Elements

We showed how to write a simple test at a high level, but haven't shown how to make individual actions in the app using Puppeteer. This was hidden in the `login()` and `logout()` functions. Let's see how `login()` is implemented.

```
// Log in using a party name and wait for the main screen to load.
const login = async (page: Page, partyName: string) => {
  const usernameInput = await page.waitForSelector(
    ".test-select-username-field",
  );
  if (usernameInput) {
    await usernameInput.click();
    await usernameInput.type(partyName);
    await page.click(".test-select-login-button");
    await page.waitForSelector(".test-select-main-menu");
  }
};
```

We first wait to receive a handle to the username input element. This is important to ensure the page and relevant elements are loaded by the time we try to act on them. We then use the element handle to click into the input and type the party name. Next we click the login button (this time assuming the button has loaded along with the rest of the page). Finally, we wait until we find we've reached the menu on the main page.

The strings used to find UI elements, e.g. `'.test-select-username-field'` and `'.test-select-login-button'`, are CSS Selectors. You may have seen them before in CSS styling of web pages. In this case we use *class selectors*, which look for CSS classes we've given to elements in our React components.

This means we must manually add classes to the components we want to test. For example, here is a snippet of the `LoginScreen` React component with classes added to the `Form` elements.

```
        <Form.Input
          fluid
          placeholder="Username"
          value={username}
          className="test-select-username-field"
          onChange={(e, { value }) => setUsername(value?.toString() ?? "")}
        />
        <Button
```

(continues on next page)

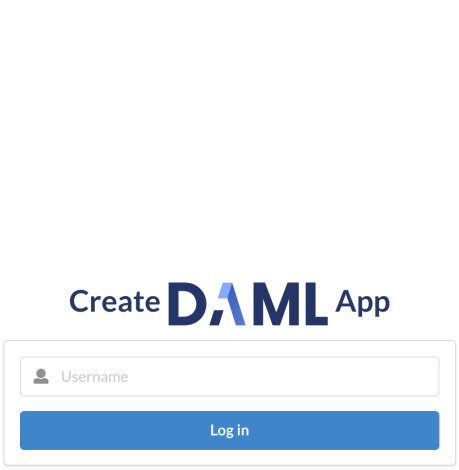
(continued from previous page)

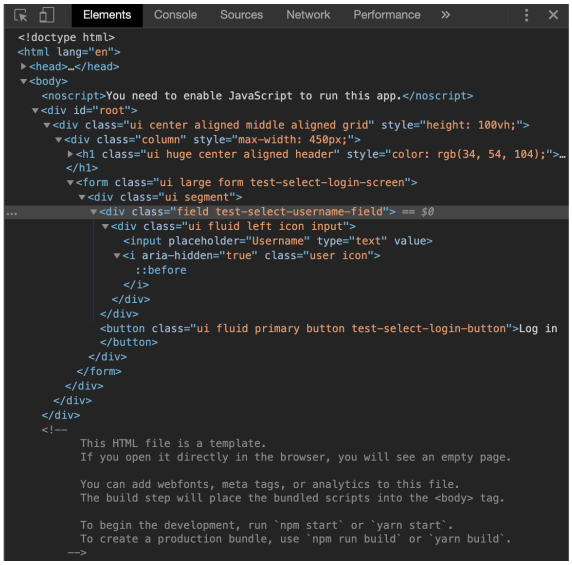
```
          primary
          fluid
          className="test-select-login-button"
          onClick={handleLogin}>
          Log in
        </Button>
```

You can see the `className` attributes in the `Input` and `Button`, which we select in the `login()` function. Note that you can use other features of an element in your selector, such as its type and attributes. We've only used class selectors in these tests.

1.9.4 Writing CSS Selectors

When writing CSS selectors for your tests, you will likely need to check the structure of the rendered HTML in your app by running it manually and inspecting elements using your browser's developer tools. For example, the image below is from inspecting the username field using the developer tools in Google Chrome.





There is a subtlety to explain here due to the Semantic UI framework we use for our app. Semantic UI provides a convenient set of UI elements which get translated to HTML. In the example of the username field above, the original Semantic UI `Input` is translated to nested `div` nodes with the `input` inside. You can see this highlighted on the right side of the screenshot. While harmless in this case, in general you may need to inspect the HTML translation of UI elements and write your CSS selectors accordingly.

1.9.5 The Full Test Suite

```
// Copyright (c) 2022 Digital Asset (Switzerland) GmbH and/or its affiliates. All
↪rights reserved.
// SPDX-License-Identifier: Apache-2.0

// Keep in sync with compatibility/bazel_tools/create-daml-app/index.test.ts

import { ChildProcess, spawn, spawnSync, SpawnOptions } from "child_process";
import { promises as fs } from "fs";
import puppeteer, { Browser, Page } from "puppeteer";
import waitOn from "wait-on";

import Ledger, { UserRightHelper, UserRight } from "@daml/ledger";
import { User } from "@daml.js/create-daml-app";
import { insecure } from "./config";

const JSON_API_PORT_FILE_NAME = "json-api.port";

const UI_PORT = 3000;

// `daml start` process
let startProc: ChildProcess | undefined = undefined;

// `npm start` process
let uiProc: ChildProcess | undefined = undefined;

// Chrome browser that we run in headless mode
let browser: Browser | undefined = undefined;

let publicUser: string | undefined;
let publicParty: string | undefined;

const adminLedger = new Ledger({
  token: insecure.makeToken("participant_admin"),
  httpBaseUrl: "http://127.0.0.1:7575/",
});

const toAlias = (userId: string): string =>
  userId.charAt(0).toUpperCase() + userId.slice(1);

// Function to generate unique party names for us.
let nextPartyId = 1;
const getParty = async (): Promise<[string, string]> => {
  const allocResult = await adminLedger.allocateParty({});
  const user = `u${nextPartyId}`;
  const party = allocResult.identifier;
  const rights: UserRight[] = [UserRightHelper.canActAs(party)].concat(
    publicParty !== undefined ? [UserRightHelper.canReadAs(publicParty)] : [],
  );
  await adminLedger.createUser(user, rights, party);
  nextPartyId++;
  return [user, party];
};

test("Party names are unique", async () => {
  let r: string[] = [];
```

(continues on next page)

(continued from previous page)

```
  for (let i = 0; i < 10; ++i) {
    r = r.concat((await getParty())[1]);
  }
  const parties = new Set(r);
  expect(parties.size).toEqual(10);
}, 20_000);

const removeFile = async (path: string) => {
  try {
    await fs.stat(path);
    await fs.unlink(path);
  } catch (_e) {
    // Do nothing if the file does not exist.
  }
};

// Start the Daml and UI processes before the tests begin.
// To reduce test times, we reuse the same processes between all the tests.
// This means we need to use a different set of parties and a new browser page
↪for each test.
beforeAll(async () => {
  // Run `daml start` from the project root (where the `daml.yaml` is located).
  // The path should include '.daml/bin' in the environment where this is run,
  // which contains the `daml` assistant executable.
  const startOpts: SpawnOptions = { cwd: "..", stdio: "inherit" };

  console.debug("Starting daml start");

  startProc = spawn("daml", ["start"], startOpts);

  await waitOn({ resources: [`tcp:127.0.0.1:6865`] });
  console.debug("daml sandbox is running");
  await waitOn({ resources: [`tcp:127.0.0.1:7575`] });
  console.debug("JSON API is running");

  [publicUser, publicParty] = await getParty();

  // Run `npm start` in another shell.
  // Disable automatically opening a browser using the env var described here:
  // https://github.com/facebook/create-react-app/issues/873#issuecomment-
↪266318338
  const env = { ...process.env, BROWSER: "none" };
  console.debug("Starting npm start");
  uiProc = spawn("npm", ["start"], {
    env,
    stdio: "inherit",
    detached: true,
  });
  // Note(kill-npm-start): The `detached` flag starts the process in a new
↪process group.
  // This allows us to kill the process with all its descendents after the tests
↪finish,
  // following https://azimi.me/2014/12/31/kill-child_process-node-js.html.

  // Ensure the UI server is ready by checking that the port is available.
  await waitOn({ resources: [`tcp:127.0.0.1:${UI_PORT}`] });
```

(continues on next page)

(continued from previous page)

```
  console.debug("npm start is running");

  // Launch a single browser for all tests.
  console.debug("Starting puppeteer");
  browser = await puppeteer.launch();
  console.debug("Puppeteer is running");
}, 60_000);

afterAll(async () => {
  // Kill the `daml start` process, allowing the sandbox and JSON API server to
  // shut down gracefully.
  // The latter process should also remove the JSON API port file.
  // TODO: Test this on Windows.
  if (startProc) {
    startProc.kill("SIGTERM");
  }

  // Kill the `npm start` process including all its descendents.
  // The `-` indicates to kill all processes in the process group.
  // See Note(kill-npm-start).
  // TODO: Test this on Windows.
  if (uiProc && uiProc.pid) {
    process.kill(-uiProc.pid);
  }

  if (browser) {
    browser.close();
  }
});

test("create and look up user using ledger library", async () => {
  const [user, party] = await getParty();
  const token = insecure.makeToken(user);
  const ledger = new Ledger({ token });
  const users0 = await ledger.query(User.User);
  expect(users0).toEqual([]);
  const userPayload = { username: party, following: [], public: publicParty };
  const userContract1 = await ledger.create(User.User, userPayload);
  const userContract2 = await ledger.fetchByKey(User.User, party);
  expect(userContract1).toEqual(userContract2);
  const users = await ledger.query(User.User);
  expect(users[0]).toEqual(userContract1);
}, 20_000);

// The tests following use the headless browser to interact with the app.
// We select the relevant DOM elements using CSS class names that we embedded
// specifically for testing.
// See https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors.

const newUiPage = async (): Promise<Page> => {
  if (!browser) {
    throw Error("Puppeteer browser has not been launched");
  }
  const page = await browser.newPage();
  await page.setViewport({ width: 1366, height: 1080 });
  page.on("console", message =>
```

(continues on next page)

(continued from previous page)

```
    console.log(
      `${message.type().substr(0, 3).toUpperCase()} ${message.text()}`,
    ),
  );
  await page.goto(`http://127.0.0.1:${UI_PORT}`); // ignore the Response
  return page;
};

// Note that Follow is a consuming choice on a contract
// with a contract key so it is crucial to wait between follows.
// Otherwise, you get errors due to contention.
// Those can manifest in puppeteer throwing `Target closed`
// but that is not the underlying error (the JSON API will
// output the contention errors as well so look through the log).
const waitForFollowers = async (page: Page, n: number) => {
  await page.waitForFunction(
    (n: number) =>
      document.querySelectorAll(".test-select-following").length == n,
    {},
    n,
  );
};

// LOGIN_FUNCTION_BEGIN
// Log in using a party name and wait for the main screen to load.
const login = async (page: Page, partyName: string) => {
  const usernameInput = await page.waitForSelector(
    ".test-select-username-field",
  );
  if (usernameInput) {
    await usernameInput.click();
    await usernameInput.type(partyName);
    await page.click(".test-select-login-button");
    await page.waitForSelector(".test-select-main-menu");
  }
};
// LOGIN_FUNCTION_END

// Log out and wait to get back to the login screen.
const logout = async (page: Page) => {
  await page.click(".test-select-log-out");
  await page.waitForSelector(".test-select-login-screen");
};

// Follow a user using the text input in the follow panel.
const follow = async (page: Page, userToFollow: string) => {
  const followInput = await page.waitForSelector(".test-select-follow-input");
  if (followInput) {
    await followInput.click();
    await followInput.type(userToFollow);
    await followInput.press("Enter");
    await page.click(".test-select-follow-button");

    // Wait for the request to complete, either successfully or after the error
    // dialog has been handled.
    // We check this by the absence of the `loading` class.
```

(continues on next page)

(continued from previous page)

```
    // (Both the `test-...` and `loading` classes appear in `div`s surrounding
    // the `input`, due to the translation of Semantic UI's `Input` element.)
    await page.waitForSelector(".test-select-follow-input > :not(.loading)", {
      timeout: 40_000,
    });
  }
};

// LOGIN_TEST_BEGIN
test("log in as a new user, log out and log back in", async () => {
  const [user, party] = await getParty();

  // Log in as a new user.
  const page = await newUiPage();
  await login(page, user);

  // Check that the ledger contains the new User contract.
  const token = insecure.makeToken(user);
  const ledger = new Ledger({ token });
  const users = await ledger.query(User.User);
  expect(users).toHaveLength(1);
  expect(users[0].payload.username).toEqual(party);

  // Log out and in again as the same user.
  await logout(page);
  await login(page, user);

  // Check we have the same one user.
  const usersFinal = await ledger.query(User.User);
  expect(usersFinal).toHaveLength(1);
  expect(usersFinal[0].payload.username).toEqual(party);

  await page.close();
}, 40_000);
// LOGIN_TEST_END

// This tests following users in a few different ways:
// - using the text box in the Follow panel
// - using the icon in the Network panel
// - while the user that is followed is logged in
// - while the user that is followed is logged out
// These are all successful cases.

test("log in as three different users and start following each other", async () =>
↪ {
  const [user1, party1] = await getParty();
  const [user2, party2] = await getParty();
  const [user3, party3] = await getParty();

  // Log in as Party 1.
  const page1 = await newUiPage();
  await login(page1, user1);

  // Log in as Party 2.
  const page2 = await newUiPage();
  await login(page2, user2);
```

(continues on next page)

(continued from previous page)

```
// Log in as Party 3.
const page3 = await newUiPage();
await login(page3, user3);

// Party 1 should initially follow no one.
const noFollowing1 = await page1.$$(".test-select-following");
expect(noFollowing1).toEqual([]);

// Follow Party 2 using the text input.
// This should work even though Party 2 has not logged in yet.
// Check Party 1 follows exactly Party 2.
await follow(page1, party2);
await waitForFollowers(page1, 1);
const followingList1 = await page1.$$eval(
  ".test-select-following",
  following => following.map(e => e.innerHTML),
);
expect(followingList1).toEqual([toAlias(user2)]);

// Add Party 3 as well and check both are in the list.
await follow(page1, party3);
await waitForFollowers(page1, 2);
const followingList11 = await page1.$$eval(
  ".test-select-following",
  following => following.map(e => e.innerHTML),
);
expect(followingList11).toHaveLength(2);
expect(followingList11).toContain(toAlias(user2));
expect(followingList11).toContain(toAlias(user3));

// Party 2 should initially follow no one.
const noFollowing2 = await page2.$$(".test-select-following");
expect(noFollowing2).toEqual([]);

// However, Party 2 should see Party 1 in the network.
await page2.waitForSelector(".test-select-user-in-network");
const network2 = await page2.$$eval(".test-select-user-in-network", users =>
  users.map(e => e.innerHTML),
);
expect(network2).toEqual([toAlias(user1)]);

// Follow Party 1 using the 'add user' icon on the right.
await page2.waitForSelector(".test-select-add-user-icon");
const userIcons = await page2.$$(".test-select-add-user-icon");
expect(userIcons).toHaveLength(1);
await userIcons[0].click();
await waitForFollowers(page2, 1);

// Also follow Party 3 using the text input.
// Note that we can also use the icon to follow Party 3 as they appear in the
// Party 1's Network panel, but that's harder to test at the
// moment because there is no loading indicator to tell when it's done.
await follow(page2, party3);

// Check the following list is updated correctly.
```

(continues on next page)

(continued from previous page)

```
  await waitForFollowers(page2, 2);
  const followingList2 = await page2.$$eval(
    ".test-select-following",
    following => following.map(e => e.innerHTML),
  );
  expect(followingList2).toHaveLength(2);
  expect(followingList2).toContain(toAlias(user1));
  expect(followingList2).toContain(toAlias(user3));

  // Party 1 should now also see Party 2 in the network (but not Party 3 as they
  // didn't yet started following Party 1).
  await page1.waitForSelector(".test-select-user-in-network");
  const network1 = await page1.$$eval(
    ".test-select-user-in-network",
    following => following.map(e => e.innerHTML),
  );
  expect(network1).toEqual([toAlias(user2)]);

  // Party 3 should follow no one.
  const noFollowing3 = await page3.$$(".test-select-following");
  expect(noFollowing3).toEqual([]);

  // However, Party 3 should see both Party 1 and Party 2 in the network.
  await page3.waitForSelector(".test-select-user-in-network");
  const network3 = await page3.$$eval(
    ".test-select-user-in-network",
    following => following.map(e => e.innerHTML),
  );
  expect(network3).toHaveLength(2);
  expect(network3).toContain(toAlias(user1));
  expect(network3).toContain(toAlias(user2));

  await page1.close();
  await page2.close();
  await page3.close();
}, 60_000);

test("error when following self", async () => {
  const [user, party] = await getParty();
  const page = await newUiPage();

  const dismissError = jest.fn(dialog => dialog.dismiss());
  page.on("dialog", dismissError);

  await login(page, user);
  await follow(page, party);

  expect(dismissError).toHaveBeenCalled();

  await page.close();
});

test("error when adding a user that you are already following", async () => {
  const [user1, party1] = await getParty();
  const [user2, party2] = await getParty();
  const page = await newUiPage();
```

(continues on next page)

(continued from previous page)

```
  const dismissError = jest.fn(dialog => dialog.dismiss());
  page.on("dialog", dismissError);

  await login(page, user1);
  // First attempt should succeed
  await follow(page, party2);
  // Second attempt should result in an error
  await follow(page, party2);

  expect(dismissError).toHaveBeenCalled();

  await page.close();
}, 10000);

const failedLogin = async (page: Page, partyName: string) => {
  let error: string | undefined = undefined;
  await page.exposeFunction("getError", () => error);
  const dismissError = jest.fn(async dialog => {
    error = dialog.message();
    await dialog.dismiss();
  });
  page.on("dialog", dismissError);
  const usernameInput = await page.waitForSelector(
    ".test-select-username-field",
  );
  if (usernameInput) {
    await usernameInput.click();
    await usernameInput.type(partyName);
    await page.click(".test-select-login-button");
    await page.waitForFunction(
      // Casting window as any so the TS compiler doesn't flag this as an
      // error.
      // The window object normally doesn't have a .getError method, but
      // we're adding one above with exposeFunction.
      async () => (await (window as any).getError()) !== undefined,
    );
    expect(dismissError).toHaveBeenCalled();
    return error;
  }
};

test("error on user id with invalid format", async () => {
  // user ids should not contains `%`
  const invalidUser = "Alice%";
  const page = await newUiPage();
  const error = await failedLogin(page, invalidUser);
  expect(error).toMatch(/User ID \\"Alice%\\" does not match regex/);
  await page.close();
}, 40_000);

test("error on non-existent user id", async () => {
  const invalidUser = "nonexistent";
  const page = await newUiPage();
  const error = await failedLogin(page, invalidUser);
  expect(error).toMatch(
```

(continues on next page)

(continued from previous page)

```
    /getting user failed for unknown user \\"nonexistent\\"/,
  );
  await page.close();
}, 40_000);

test("error on user with no primary party", async () => {
  const invalidUser = "noprimary";
  await adminLedger.createUser(invalidUser, []);
  const page = await newUiPage();
  const error = await failedLogin(page, invalidUser);
  expect(error).toMatch(/User 'noprimary' has no primary party/);
  await page.close();
}, 40_000);
```

1.10 Overview: Important Considerations When Building Applications With Daml

1.10.1 Overall Considerations

Because Daml provides a unique and innovative solution to the problem of multi-party applications, some of the common architectural approaches used in existing solutions do not apply when working with Daml. You must understand Daml's architecture and principles and design your application and deployment approaches accordingly.

Canton is fast and highly scalable, but it performs differently than traditional databases, particularly those that follow a monolithic architecture. Transactions are processed in fractions of a second - quite fast for a distributed ledger (the blockchains used in cryptocurrencies like Bitcoin or Ethereum take many minutes to complete transactions) but slower than most traditional databases due to its distributed nature. Application design must take this into account.

Each component of Daml can be scaled, including running multiple domains and domain nodes, multiple participant nodes, and multiple parties. Integration components, e.g. HTTP JSON API Service and Trigger Service, also scale. Some components require that data is sharded in order to scale.

1.10.2 Developer Considerations

When programming within a distributed system like Daml, the developer must view every action of the system as an asynchronous operation; contention is natural and expected. This contention can stifle the performance of applications if not handled properly. The aim is to reduce contention and handle it gracefully, not to eliminate it at all costs. If contention only occurs rarely, it may be cheaper in terms of both performance and complexity to let the occasional allocation fail and retry it than to implement sharding or other complex processes.

Application design must understand the sources of contention; this allows you to use different techniques to manage it and improve performance by increasing throughput and decreasing latency. These techniques include:

> Bundling or batching business logic to increase business transaction throughput - the marginal cost of extra business logic within a transaction is often small, so bundling or batching business logic cleverly can allow for throughput an order of magnitude higher.

Maximizing parallelism with techniques like sharding, ensuring there is no contention between shards.

When designing Daml applications:

Understand where contention occurs
Split contracts across natural lines to reduce single high contention contracts (e.g., don't represent asset holdings for all owners and types as a dictionary on a single contract, but as individual contracts)
Partition contracts along natural lines and touch as few partitions as possible in each transaction (e.g., partition all asset positions into total asset positions, and then only touch one total asset position per transaction)
Use contention-free representations where possible

For more information, see the section on Avoiding Contention.

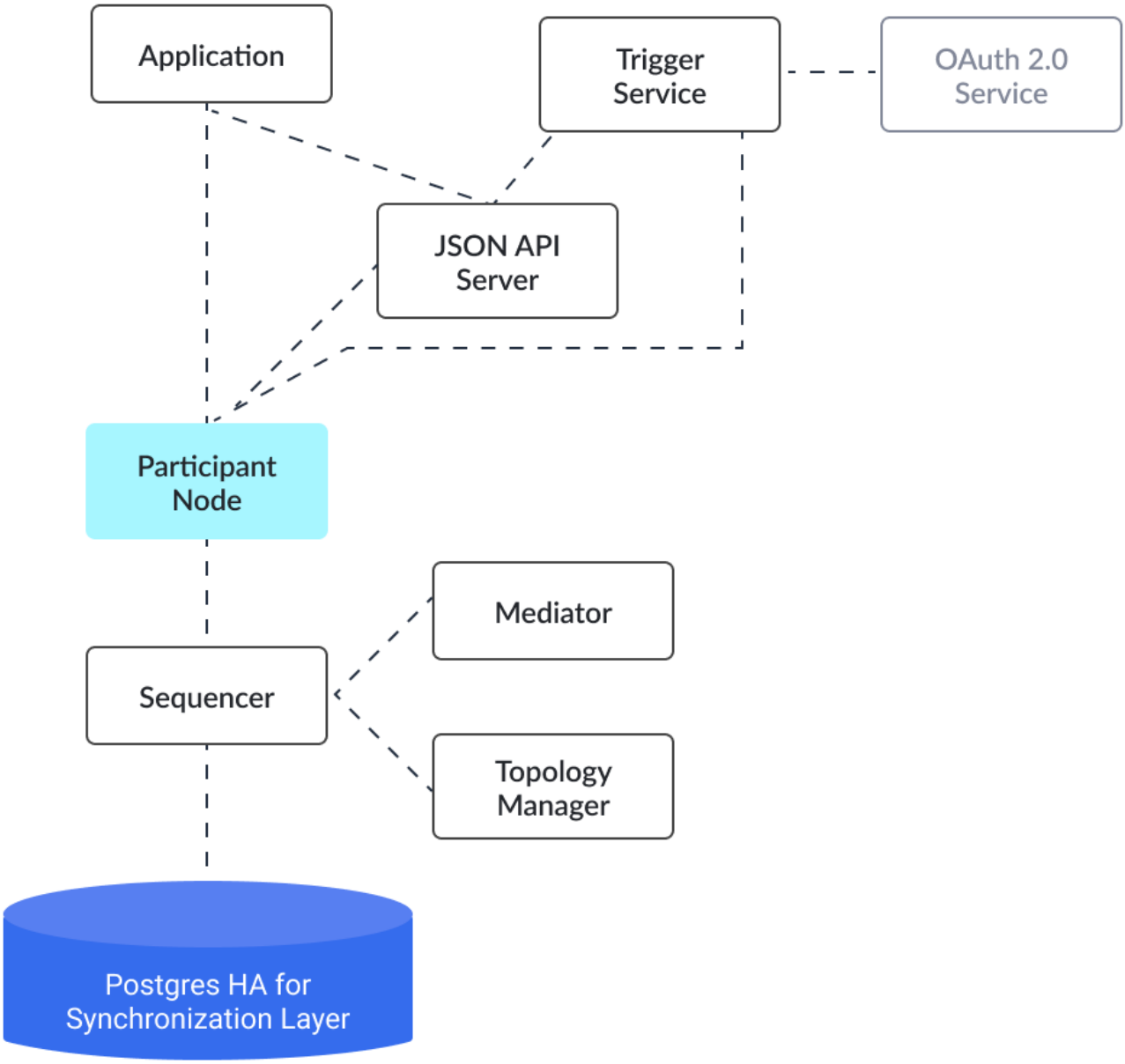
The Daml language follows functional programming principles. To build safe, secure smart contracts with Daml, we recommend that the developers embrace functional programming.

The Daml SDK contains tools and libraries that simplify multi-party application development, including defining the application's schema and implementing off-ledger code that leverages the Canton APIs.

1.10.3 Operational Considerations

Most components of Daml store state, so deployment techniques that follow stateless practices can be problematic within Daml. Achieving high availability and scalability requires clear understanding of the purpose of each component within the Daml solution. While all components in Daml scale horizontally, stateful components (e.g. participant nodes) scale horizontally via sharding.

The diagram below shows the components often used in a Daml deployment. High availability is achieved via either active-active (HTTP JSON API Service, sequencer) or active-passive (participant node, mediator) clustering. Node scaling is achieved via horizontal scaling with participant nodes requiring sharding across participants.



1.10.4 Next Steps

Go to *An Introduction to Daml* to begin learning how to write smart contracts with Daml.

1.11 Write Smart Contracts with Daml

1.11.1 An Introduction to Daml

Daml is a smart contract language designed to build composable applications on an abstract *Daml Ledger Model*.

In this introduction, you will learn about the structure of a Daml Ledger, and how to write Daml applications that run on any Daml Ledger implementation, by building an asset-holding and -trading

application. You will gain an overview over most important language features, how they relate to the *Daml Ledger Model* and how to use Daml's developer tools to write, test, compile, package and ship your application.

This introduction is structured such that each section presents a new self-contained application with more functionality than that from the previous section. You can find the Daml code for each section here or download them using the Daml assistant. For example, to load the sources for section 1 into a folder called `intro1`, run `daml new intro1 --template daml-intro-1`.

Prerequisites:

> You have installed the *Daml SDK*

Next: *Basic Contracts*.

1.11.2 Basic Contracts

To begin with, you're going to write a very small Daml template, which represents a self-issued, non-transferable token. Because it's a minimal template, it isn't actually useful on its own - you'll make it more useful later - but it's enough that it can show you the most basic concepts:

> Transactions
> Daml Modules and Files
> Templates
> Contracts
> Signatories

Hint: Remember that you can load all the code for this section into a folder `intro1` by running `daml new intro1 --template daml-intro-1`

1.11.2.1 Daml Ledger Basics

Like most structures called ledgers, a Daml Ledger is just a list of *commits*. When we say *commit*, we mean the final result of when a *party* successfully *submits* a *transaction* to the ledger.

Transaction is a concept we'll cover in more detail through this introduction. The most basic examples are the creation and archival of a *contract*.

A contract is *active* from the point where there is a committed transaction that creates it, up to the point where there is a committed transaction that *archives* it.

Individual contracts are *immutable* in the sense that an active contract can not be changed. You can only change the *active contract set* by creating a new contract, or archiving an old one.

Daml specifies what transactions are legal on a Daml Ledger. The rules the Daml code specifies are collectively called a *Daml model* or *contract model*.

1.11.2.2 Daml Files and Modules

Each `.daml` file defines a *Daml Module* at the top:

```
module Token where
```

Code comments in Daml are introduced with `--`:

```
-- A Daml file defines a module.
module Token where
```

1.11.2.3 Templates

A `template` defines a type of contract that can be created, and who has the right to do so. *Contracts* are instances of *templates*.

Listing 1: A simple template

```
template Token
  with
    owner : Party
  where
    signatory owner
```

You declare a template starting with the `template` keyword, which takes a name as an argument.

Daml is whitespace-aware and uses layout to structure *blocks*. Everything that's below the first line is indented, and thus part of the template's body.

Contracts contain data, referred to as the *create arguments* or simply *arguments*. The `with` block defines the data type of the create arguments by listing field names and their types. The single colon `:` means of type , so you can read this as template `Token` with a field `owner` of type `Party` .

`Token` contracts have a single field `owner` of type `Party`. The fields declared in a template's `with` block are in scope in the rest of the template body, which is contained in a `where` block.

1.11.2.4 Signatories

The `signatory` keyword specifies the *signatories* of a contract. These are the parties whose *authority* is required to create the contract or archive it – just like a real contract. Every contract must have at least one signatory.

Furthermore, Daml ledgers *guarantee* that parties see all transactions where their authority is used. This means that signatories of a contract are guaranteed to see the creation and archival of that contract.

1.11.2.5 Next Up

In *Test Templates Using Daml Script*, you'll learn about how to try out the `Token` contract template in Daml's inbuilt Daml Script testing language.

1.11.3 Test Templates Using Daml Script

In this section we test the `Token` model from *Basic Contracts* using the *Daml Script* integration in *Daml Studio*. This includes:

Script basics
Running scripts
Creating contracts
Testing for failure
Archiving contracts
Viewing the ledger and ledger history

Hint: Remember that you can load all the code for this section into a folder called `intro2` by running `daml new intro2 --template daml-intro-2`

1.11.3.1 Script Basics

A `Script` is like a recipe for a test, letting you create a scenario where different parties submit a series of transactions to check that your templates behave as you expect. You can also script some external information like party identities and ledger time.

Below is a basic script that creates a `Token` for a party called Alice :

```
token_test_1 = script do
  alice <- allocateParty "Alice"
  submit alice do
    createCmd Token with owner = alice
```

You declare a `Script` as a top-level variable and introduce it using `script do`. `do` always starts a block, so the rest of the script is indented.

Before you can create any `Token` contracts, you need some parties on the test ledger. The above script uses the function `allocateParty` to put a party called Alice in a variable `alice`. There are two things of note there:

Use of <- instead of =.
The reason for this is that `allocateParty` is an `Action` that can only be performed once the `Script` is run in the context of a ledger. <- means run the action and bind the result . It can only be run in that context because, depending on the ledger state, `allocateParty` gives you back a party with the name you specified or appends a suffix to that name if such a party has already been allocated. You can read more about `Actions` and `do` blocks in *Add Constraints to a Contract*.
If that doesn't quite make sense yet, for the time being you can think of this arrow as extracting the right-hand-side value from the ledger and storing it into the variable on the left.
The argument `"Alice"` to `allocateParty` does not have to be enclosed in brackets. Functions in Daml are called using the syntax `fn arg1 arg2 arg3`.

With a variable `alice` of type `Party` in hand, you can submit your first transaction using the `submit` function. `submit` takes two arguments: the `Party` and the `Commands`.

Just like `Script` is a recipe for a test, `Commands` is a recipe for a transaction. `createCmd Token with owner = alice` is a `Commands`, which translates to a list of commands to be submitted to the ledger. These commands create a transaction which in turns creates a `Token` with owner Alice.

You'll learn all about the syntax `Token with owner = alice` in *Data Types*.

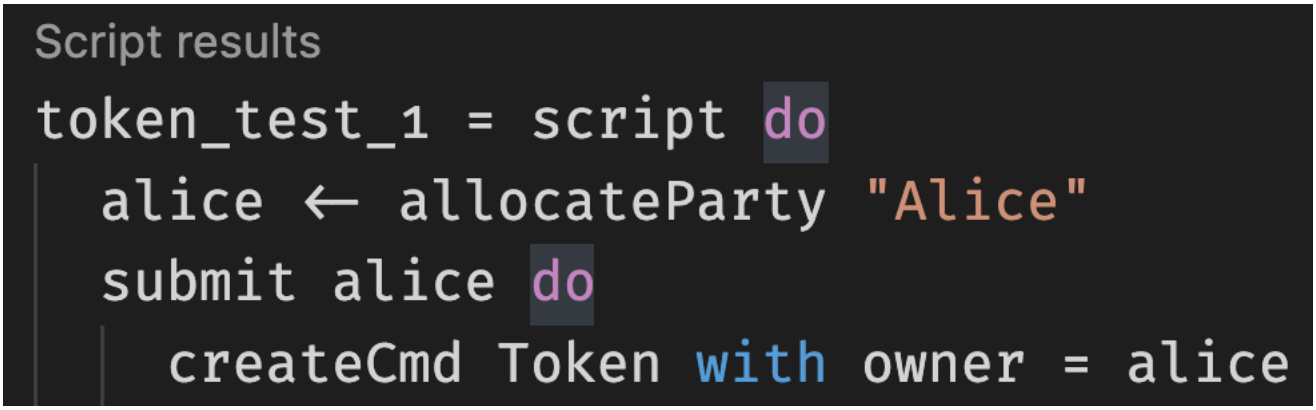
You could write this as `submit alice (createCmd Token with owner = alice)`, but as with scripts, you can assemble commands using `do` blocks. A `do` block always takes the value of the last statement within it so the syntax shown in the commands above gives the same result, whilst being easier to read. Note, however, that the commands submitted as part of a transaction are not allowed to depend on each other.

1.11.3.2 Run the Scripts

There are a few ways to run Daml Scripts:

In Daml Studio against a test ledger, providing visualizations of the resulting ledger.
Using the command line `daml test` also against a test ledger, useful for continuous integration.
Against a real ledger. See the documentation for *Daml Script* for more information.
Interactively using *Daml REPL*.

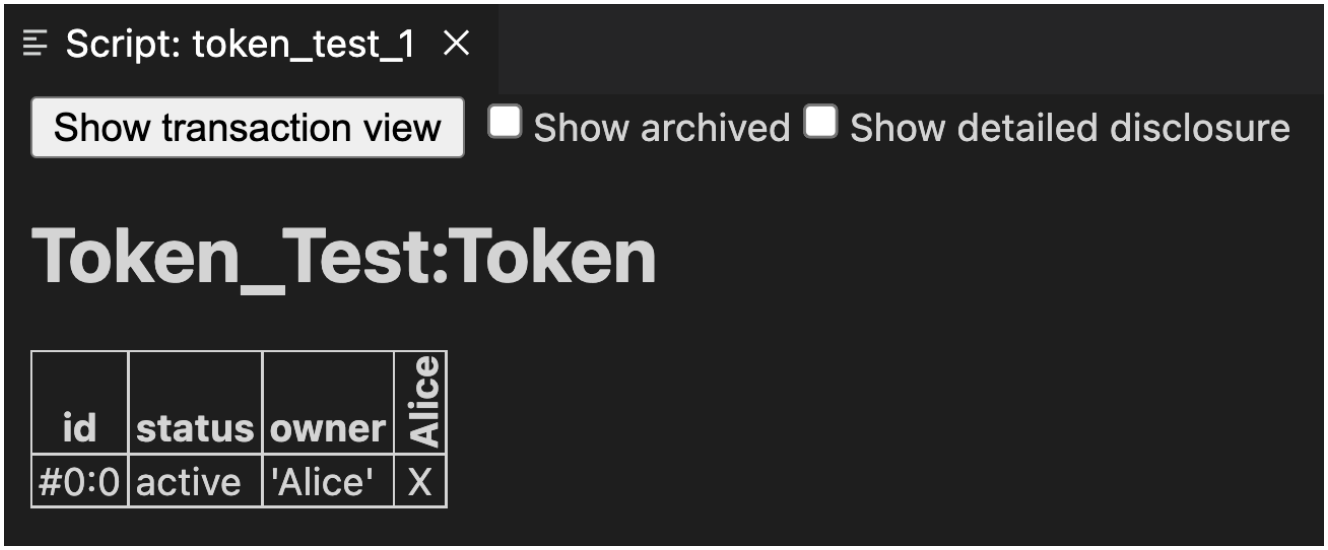
In Daml Studio, you should see the text Script results just above the line `token_test_1 = do`. Click on that text to display the outcome of the script.



This opens the script view in a separate column in VS Code. The default view is a tabular representation of the final state of the ledger:

What this display means:

The big title reading `Token_Test:Token` identifies the type of contract that's listed below. `Token_Test` is the module name, `Token` is the template name.
The first column shows the ID of the contract. This will be explained later.
The second column shows the status of the contract, either `active` or `archived`.
The next section of columns show the contract arguments, with one column per field. As expected, here there is one field and thus one column: the field `owner` is `'Alice'`. The single quotation marks indicate that `Alice` is a party.
The remaining columns, labelled vertically, show which parties know about which contracts. In this simple script, the sole party Alice knows about the contract she created.



To run the same test from the command line, save your module in a file `Token_Test.daml` and run `daml test --files Token_Test.daml`. If your file contains more than one script, this runs all of them.

1.11.3.3 Test for Failure

In *Basic Contracts* you learned that creating a `Token` requires the authority of its owner. In other words, it should not be possible for Alice to create a token for another party, e.g. Bob, or vice versa. A reasonable attempt to test that would be:

```
failing_test_1 = do
  alice <- allocateParty "Alice"
  bob <- allocateParty "Bob"
  submit alice do
    createCmd Token with owner = bob
  submit bob do
    createCmd Token with owner = alice
```

However, if you open the script view for that script, you see the following message:

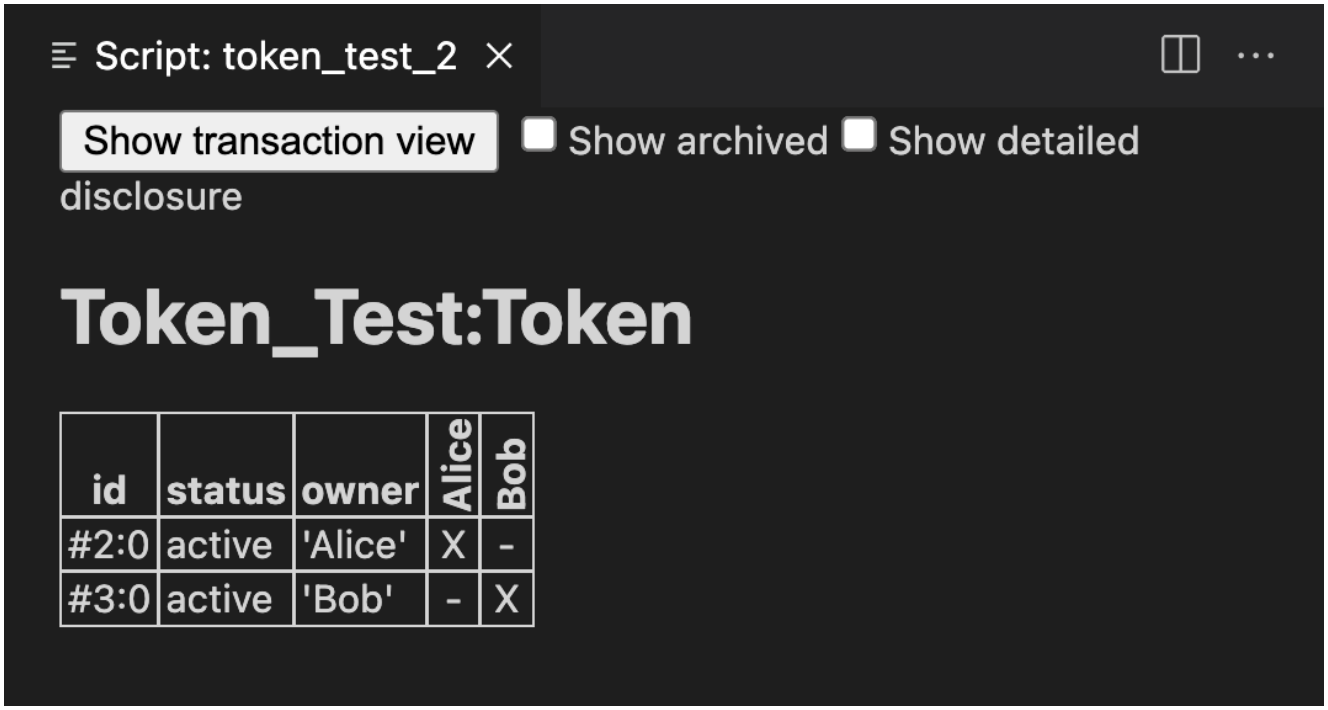
```
Script execution failed on commit at Token_Test:64:3:
  0: create of Token_Test:Token at DA.Internal.Template.Functions:229:3
     failed due to a missing authorization from 'Bob'

Ledger time: 1970-01-01T00:00:00Z

Partial transaction:
  Sub-transactions:
     0
     └─> 'Bob' creates Token_Test:Token
               with
                 owner = 'Bob'
```

The script failed, as expected, but scripts abort at the first failure. This means that it only tested that Alice cannot create a token for Bob; the second `submit` statement was never reached.

To test for failing submits and keep the script running thereafter, or fail if the submission succeeds, you can use the `submitMustFail` function:

```
token_test_2 = do
  alice <- allocateParty "Alice"
  bob <- allocateParty "Bob"

  submitMustFail alice do
    createCmd Token with owner = bob
  submitMustFail bob do
    createCmd Token with owner = alice

  submit alice do
    createCmd Token with owner = alice
  submit bob do
    createCmd Token with owner = bob
```

`submitMustFail` never has an impact on the ledger, so the resulting tabular script view only shows the two tokens resulting from the successful `submit` statements. Note the new column for Bob as well as the visibilities. Alice and Bob cannot see each others' tokens.



1.11.3.4 Archive Contracts

Archiving contracts is the counterpart to creating contracts. Contracts are immutable, so whenever you want to update one (loosely: change its state) you must archive the current contract residing on the ledger and create a new one.

To archive a contract, use `archiveCmd` instead of `createCmd`. Whereas `createCmd` takes an instance of a template, `archiveCmd` takes a reference to a created contract. Archiving requires authorization from controllers.

Contracts are also archived whenever a consuming choice is exercised.

Important: Archive choices are present on all templates and cannot be removed.

References to contracts have the type `ContractId a`, where `a` is a *type parameter* representing the template type of the contract that the id refers to. For example, a reference to a `Token` would be a `ContractId Token`.

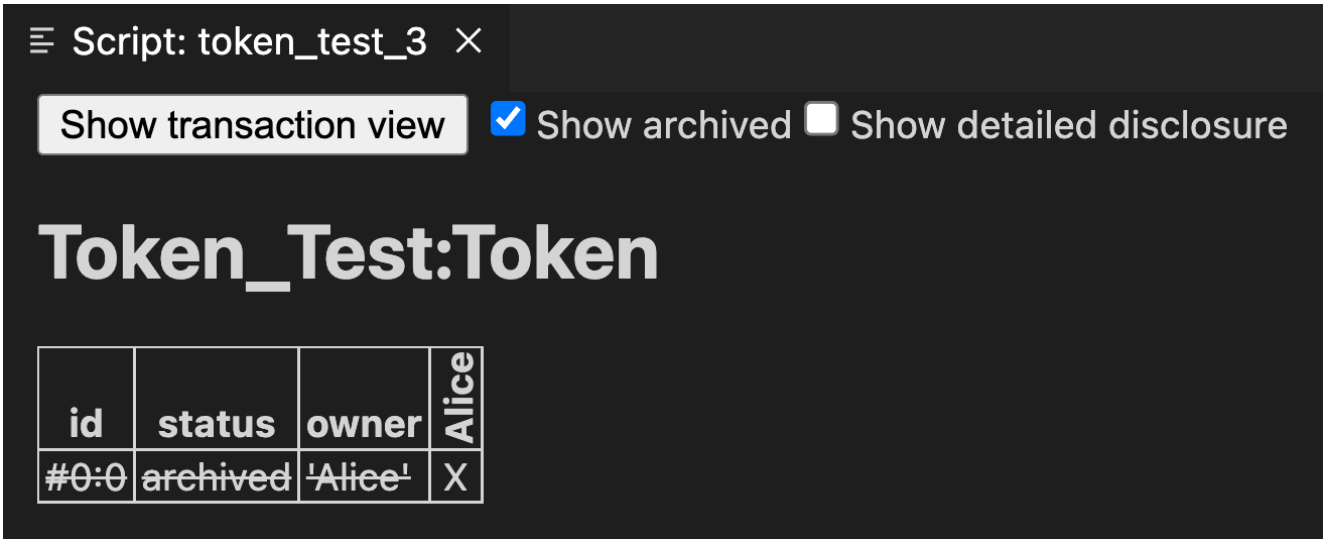
To `archiveCmd` the token Alice has created, you need the contract id. Retrieve the contract id from the ledger with the `<-` notation. How this works is discussed in *Add Constraints to a Contract*.

This script first checks that Bob cannot archive Alice's token. Then Alice successfully archives it:

```
token_test_3 = do
  alice <- allocateParty "Alice"
  bob <- allocateParty "Bob"

  alice_token <- submit alice do
    createCmd Token with owner = alice

  submitMustFail bob do
    archiveCmd alice_token

  submit alice do
    archiveCmd alice_token
```

1.11.3.5 View the Ledger and Ledger History

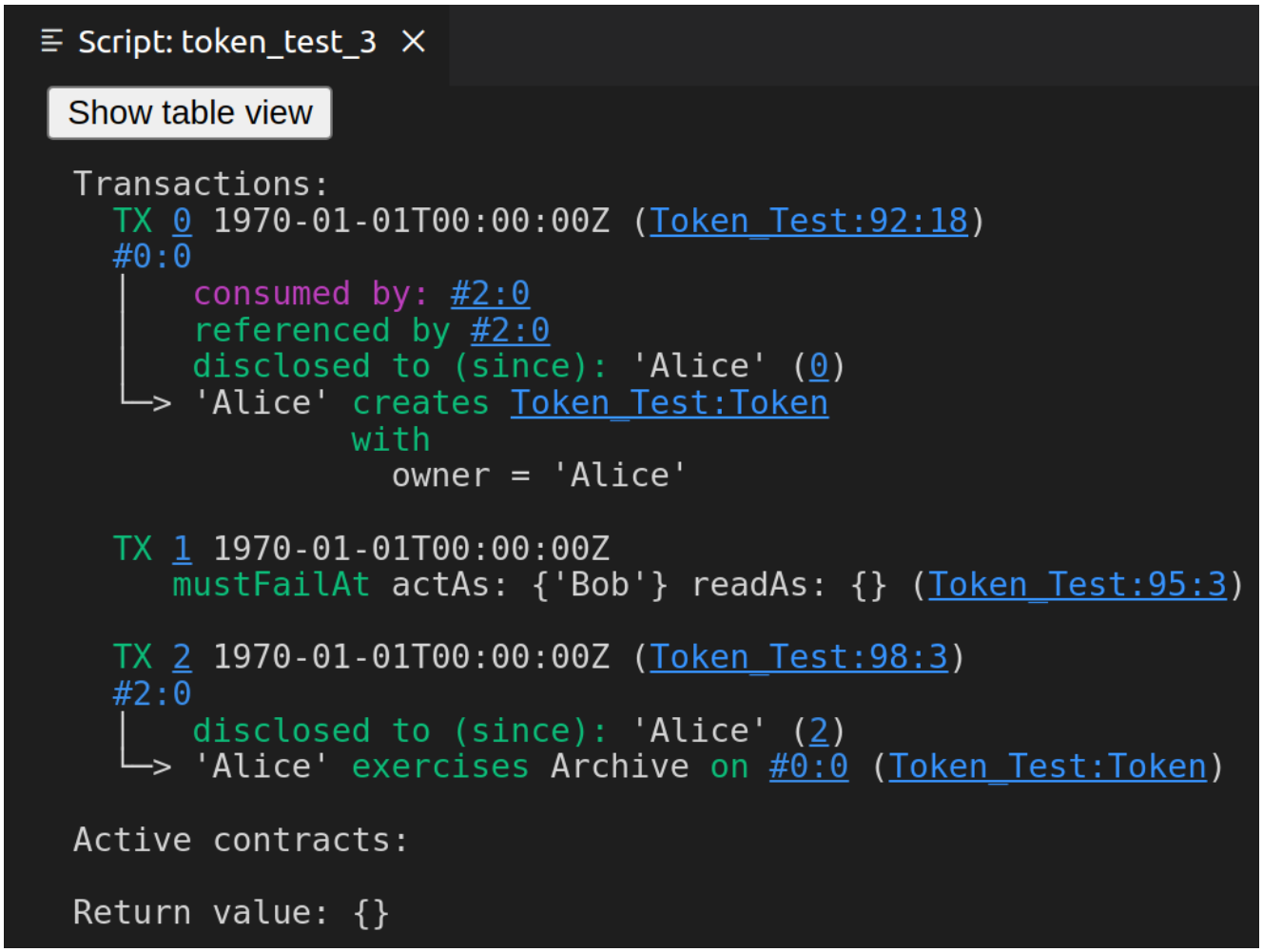
Once you archive the contract the resulting script view is empty; there are no contracts left on the ledger. If you want to see the history of the ledger, e.g. to see how you got to that state, tick the Show archived box at the top of the ledger view:



You can see that there was a `Token` contract, which is now archived, indicated both by the archived value in the `status` column as well as by a strikethrough.

Click on the adjacent Show transaction view button to see the entire transaction graph:

In the Daml Studio script runner, committed transactions are numbered sequentially. In the image above, the lines starting with `TX` indicate that there are three committed transactions, with ids `#0`,


Script: token_test_3
Show table view
Transactions:
TX 0 1970-01-01T00:00:00Z (Token_Test:92:18)
#0:0
consumed by: #2:0
referenced by #2:0
disclosed to (since): 'Alice' (0)
'Alice' creates Token_Test:Token
with
owner = 'Alice'
TX 1 1970-01-01T00:00:00Z
mustFailAt actAs: {'Bob'} readAs: {} (Token_Test:95:3)
TX 2 1970-01-01T00:00:00Z (Token_Test:98:3)
#2:0
disclosed to (since): 'Alice' (2)
'Alice' exercises Archive on #0:0 (Token_Test:Token)
Active contracts:
Return value: {}

`#1`, and `#2`. These correspond to the three `submit` and `submitMustFail` statements in the script.

Transaction `#0` has one *sub-transaction* `#0:0`, which the arrow indicates is a `create` of a `Token`. Identifiers `#X:Y` mean `commit X, sub-transaction Y`. All transactions have this format in the script runner. However, this format is a testing feature. In general, you should consider Transaction and Contract IDs to be opaque.

The lines above and below `create Token_Test:Token` give additional information:

`consumed by: #2:0` tells you that the contract is archived in sub-transaction `0` of commit `2`.
`referenced by #2:0` tells you that the contract was used in other transactions, and lists their IDs.
`disclosed to (since): 'Alice' (#0)` tells you who knows about the contract. The fact that `'Alice'` appears in the list is equivalent to an `x` in the tabular view. The `(#0)` gives you the additional information that `Alice` learned about the contract in commit `#0`.
Everything following `with` shows the create arguments.

1.11.3.6 Exercises

To get a better understanding of script, try the following exercises:

1. Write a template for a second type of token.
2. Write a script with two parties and two types of tokens, creating one token of each type for each party and archiving one token for each party, leaving one token of each type in the final ledger view.
3. In *Archive Contracts* you tested that Bob cannot archive Alice's token. Can you guess why the submit fails? How can you find out why the submit fails?

Hint: Remember that in *Test for Failure* we saw a proper error message for a failing submit.

1.11.3.7 Next Up

In *Data Types* you will learn about Daml's type system, and how you can think of templates as tables and contracts as database rows.

1.11.4 Data Types

In *Basic Contracts*, you learnt about contract templates, which specify the types of contracts that can be created on the ledger, and what data those contracts hold in their arguments.

In *Test Templates Using Daml Script*, you learnt about the script view in Daml Studio, which displays the current ledger state. It shows one table per template, with one row per contract of that type and one column per field in the arguments.

This actually provides a useful way of thinking about templates: like tables in databases. Templates specify a data schema for the ledger:

each template corresponds to a table
each field in the `with` block of a template corresponds to a column in that table
each contract of that type corresponds to a table row

In this section, you'll learn how to create rich data schemas for your ledger. Specifically you'll learn about:

Daml's built-in and native data types
Record types
Derivation of standard properties
Variants
Manipulating immutable data
Contract keys

After this section, you should be able to use a Daml ledger as a simple database where individual parties can write, read and delete complex data.

Hint: Remember that you can load all the code for this section into a folder called `intro3` by running `daml new intro3 --template daml-intro-3`

1.11.4.1 Native Types

You have already encountered a few native Daml types: `Party` in *Basic Contracts*, and `Text` and `ContractId` in *Test Templates Using Daml Script*. Here are those native types and more:

`Party` Stores the identity of an entity that is able to act on the ledger, in the sense that they can sign contracts and submit transactions. In general, `Party` is opaque.
`Text` Stores a unicode character string like `"Alice"`.
`ContractId a` Stores a reference to a contract of type `a`.
`Int` Stores signed 64-bit integers. For example, `-123`.
`Decimal` Stores fixed-point number with 28 digits before and 10 digits after the decimal point. For example, `0.0000000001` or `-99999999999999999999999999.9999999999`.
`Bool` Stores `True` or `False`.
`Date` Stores a date.
`Time` Stores absolute UTC time.
`RelTime` Stores a difference in time.

The below script instantiates each one of these types, manipulates it where appropriate, and tests the result:

```
import Daml.Script
import DA.Time
import DA.Date

native_test = script do
  alice <- allocateParty "Alice"
  bob <- allocateParty "Bob"
  let
    my_int = -123
    my_dec = 0.001 : Decimal
    my_text = "Alice"
    my_bool = False
    my_date = date 2020 Jan 01
    my_time = time my_date 00 00 00
    my_rel_time = hours 24
```

(continues on next page)

(continued from previous page)

```
  assert (alice /= bob)
  assert (-my_int == 123)
  assert (1000.0 * my_dec == 1.0)
  assert (my_text == "Alice")
  assert (not my_bool)
  assert (addDays my_date 1 == date 2020 Jan 02)
  assert (addRelTime my_time my_rel_time == time (addDays my_date 1) 00 00 00)
```

Despite its simplicity, there are quite a few things to note in this script:

The `import` statements at the top import two packages from the Daml Standard Library, which contain all the date and time related functions we use here as well as the functions used in Daml Scripts. More on packages, imports and the standard library later.
Most of the variables are declared inside a `let` block.
That's because the `script do` block expects script actions like `submit` or `allocateParty`. An integer like `123` is not an action, it's a pure expression, something we can evaluate without any ledger. You can think of the `let` as turning variable declaration into an action.
Most variables do not have annotations to say what type they are.
That's because Daml is very good at *inferring* types. The compiler knows that `123` is an `Int`, so if you declare `my_int = 123`, it can infer that `my_int` is also an `Int`. This means you don't have to write the type annotation `my_int : Int = 123`.
However, if the type is ambiguous so that the compiler can't infer it, you do have to add a type annotation. This is the case for `0.001` which could be any `Numeric n`. Here we specify `0.001 : Decimal` which is a synonym for `Numeric 10`. You can always choose to add type annotations to aid readability.
The `assert` function is an action that takes a boolean value and succeeds with `True` and fails with `False`.
Try putting `assert False` somewhere in a script and see what happens to the script result.

With templates and these native types, it's already possible to write a schema akin to a table in a relational database. Below, `Token` is extended into a simple `CashBalance`, administered by a party in the role of an accountant:

```
template CashBalance
  with
    accountant : Party
    currency : Text
    amount : Decimal
    owner : Party
    account_number : Text
    bank : Party
    bank_address : Text
    bank_telephone : Text
  where
    signatory accountant

cash_balance_test = script do
  accountant <- allocateParty "Bob"
  alice <- allocateParty "Alice"
  bob <- allocateParty "Bank of Bob"

  submit accountant do
    createCmd CashBalance with
      accountant
```

(continues on next page)

(continued from previous page)

```
      currency = "USD"
      amount = 100.0
      owner = alice
      account_number = "ABC123"
      bank = bob
      bank_address = "High Street"
      bank_telephone = "012 3456 789"
```

1.11.4.2 Assemble Types

There's quite a lot of information on the `CashBalance` above and it would be nice to be able to give that data more structure. Fortunately, Daml's type system has a number of ways to assemble these native types into much more expressive structures.

Tuples

A common task is to group values in a generic way. Take, for example, a key-value pair with a `Text` key and an `Int` value. In Daml, you could use a two-tuple of type `(Text, Int)` to do so. If you wanted to express a coordinate in three dimensions, you could group three `Decimal` values using a three-tuple `(Decimal, Decimal, Decimal)`:

```
import DA.Tuple
import Daml.Script

tuple_test = script do
  let
    my_key_value = ("Key", 1)
    my_coordinate = (1.0 : Decimal, 2.0 : Decimal, 3.0 : Decimal)

  assert (fst my_key_value == "Key")
  assert (snd my_key_value == 1)
  assert (my_key_value._1 == "Key")
  assert (my_key_value._2 == 1)

  assert (my_coordinate == (fst3 my_coordinate, snd3 my_coordinate, thd3 my_
↪coordinate))
  assert (my_coordinate == (my_coordinate._1, my_coordinate._2, my_coordinate._3))
```

You can access the data in the tuples using:

- functions `fst`, `snd`, `fst3`, `snd3`, `thd3`
- a dot-syntax with field names `_1`, `_2`, `_3`, etc.

Daml supports tuples with up to 20 elements, but accessor functions like `fst` are only included for 2- and 3-tuples.

Lists

Lists in Daml take a single type parameter defining the type of thing in the list. So you can have a list of integers `[Int]` or a list of strings `[Text]`, but not a list mixing integers and strings.

That's because Daml is statically and strongly typed. When you get an element out of a list, the compiler needs to know what type that element has.

The below script instantiates a few lists of integers and demonstrates the most important list functions.

```
import DA.List
import Daml.Script

list_test = script do
  let
    empty : [Int] = []
    one = [1]
    two = [2]
    many = [3, 4, 5]

  -- `head` gets the first element of a list
  assert (head one == 1)
  assert (head many == 3)

  -- `tail` gets the remainder after head
  assert (tail one == empty)
  assert (tail many == [4, 5])

  -- `++` concatenates lists
  assert (one ++ two ++ many == [1, 2, 3, 4, 5])
  assert (empty ++ many ++ empty == many)

  -- `::` adds an element to the beginning of a list.
  assert (1 :: 2 :: 3 :: 4 :: 5 :: empty == 1 :: 2 :: many)
```

Note the type annotation on `empty : [Int] = []`. It's necessary because `[]` is ambiguous. It could be a list of integers or of strings, but the compiler needs to know which it is.

Records

You can think of records as named tuples with named fields. Declare them using the `data` keyword: `data T = C with`, where `T` is the type name and `C` is the data constructor. In practice, it's a good idea to always use the same name for type and data constructor:

```
data MyRecord = MyRecord with
  my_txt : Text
  my_int : Int
  my_dec : Decimal
  my_list : [Text]

-- Fields of same type can be declared in one line
data Coordinate = Coordinate with
  x, y, z : Decimal
```

(continues on next page)

(continued from previous page)

```
-- Custom data types can also have variables
data KeyValue k v = KeyValue with
  my_key : k
  my_val : v

data Nested = Nested with
  my_coord : Coordinate
  my_record : MyRecord
  my_kv : KeyValue Text Int

record_test = script do
  let
    my_record = MyRecord with
      my_txt = "Text"
      my_int = 2
      my_dec = 2.5
      my_list = ["One", "Two", "Three"]

    my_coord = Coordinate with
      x = 1.0
      y = 2.0
      z = 3.0

    -- `my_text_int` has type `KeyValue Text Int`
    my_text_int = KeyValue with
      my_key = "Key"
      my_val = 1

    -- `my_int_decimal` has type `KeyValue Int Decimal`
    my_int_decimal = KeyValue with
      my_key = 2
      my_val = 2.0 : Decimal

    -- If variables are in scope that match field names, we can pick them up
    -- implicitly, writing just `my_coord` instead of `my_coord = my_coord`.
    my_nested = Nested with
      my_coord
      my_record
      my_kv = my_text_int

  -- Fields can be accessed with dot syntax
  assert (my_coord.x == 1.0)
  assert (my_text_int.my_key == "Key")
  assert (my_nested.my_record.my_dec == 2.5)
```

You'll notice that the syntax to declare records is very similar to the syntax used to declare templates. That's no accident because a template is really just a special record. When you write `template Token with`, one of the things that happens in the background is that this becomes a `data Token = Token with`.

In the `assert` statements above, we always compared values of in-built types. If you wrote `assert (my_record == my_record)` in the script, you may be surprised to get an error message `No instance for (Eq MyRecord) arising from a use of '=='`. Equality in Daml is always value equality and we haven't written a function to check value equality for `MyRecord` values. But don't worry, you don't have to implement this rather obvious function yourself. The compiler is smart enough to do it for you, if you use `deriving (Eq)`:

```
data EqRecord = EqRecord with
  my_txt : Text
  my_int : Int
  my_dec : Decimal
  my_list : [Text]
    deriving (Eq)

data MyContainer a = MyContainer with
  contents : a
    deriving (Eq)

eq_test = script do
  let
    eq_record = EqRecord with
      my_txt = "Text"
      my_int = 2
      my_dec = 2.5
      my_list = ["One", "Two", "Three"]

    my_container = MyContainer with
      contents = eq_record
    other_container = MyContainer with
      contents = eq_record

  assert(my_container.contents == eq_record)
  assert(my_container == other_container)
```

`Eq` is what is called a *typeclass*. You can think of a typeclass as being like an interface in other languages: it is the mechanism by which you can define a set of functions (for example, `==` and `/=` in the case of `Eq`) to work on multiple types, with a specific implementation for each type they can apply to.

There are some other typeclasses that the compiler can derive automatically. Most prominently, `Show` to get access to the function `show` (equivalent to `toString` in many languages) and `Ord`, which gives access to comparison operators <, >, <=, >=.

It's a good idea to always derive `Eq` and `Show` using `deriving (Eq, Show)`. The record types created using `template T with` do this automatically, and the native types have appropriate typeclass instances. Eg `Int` derives `Eq`, `Show` and `Ord`, and `ContractId a` derives `Eq` and `Show`.

Records can give the data on `CashBalance` a bit more structure:

```
data Bank = Bank with
  party : Party
  address: Text
  telephone : Text
    deriving (Eq, Show)

data Account = Account with
  owner : Party
  number : Text
  bank : Bank
    deriving (Eq, Show)

data Cash = Cash with
  currency : Text
```

(continues on next page)

(continued from previous page)

```
  amount : Decimal
    deriving (Eq, Show)

template CashBalance
  with
    accountant : Party
    cash : Cash
    account : Account
  where
    signatory accountant

cash_balance_test = script do
  accountant <- allocateParty "Bob"
  owner <- allocateParty "Alice"
  bank_party <- allocateParty "Bank"
  let
    bank = Bank with
      party = bank_party
      address = "High Street"
      telephone = "012 3456 789"
    account = Account with
      owner
      bank
      number = "ABC123"
    cash = Cash with
      currency = "USD"
      amount = 100.0

  submit accountant do
    createCmd CashBalance with
      accountant
      cash
      account
  pure ()
```

If you look at the resulting script view, you'll see that this still gives rise to one table. The records are expanded out into columns using dot notation.

Variants and Pattern Matching

Suppose now that you also wanted to keep track of cash in hand. Cash in hand doesn't have a bank, but you can't just leave `bank` empty. Daml doesn't have an equivalent to `null`. Variants can express that cash can either be in hand or at a bank:

```
data Bank = Bank with
  party : Party
  address: Text
  telephone : Text
    deriving (Eq, Show)

data Account = Account with
  number : Text
  bank : Bank
    deriving (Eq, Show)
```

(continues on next page)

(continued from previous page)

```
data Cash = Cash with
  currency : Text
  amount : Decimal
    deriving (Eq, Show)

data Location
  = InHand
  | InAccount Account
    deriving (Eq, Show)

template CashBalance
  with
    accountant : Party
    owner : Party
    cash : Cash
    location : Location
  where
    signatory accountant

cash_balance_test = do
  accountant <- allocateParty "Bob"
  owner <- allocateParty "Alice"
  bank_party <- allocateParty "Bank"
  let
    bank = Bank with
      party = bank_party
      address = "High Street"
      telephone = "012 3456 789"
    account = Account with
      bank
      number = "ABC123"
    cash = Cash with
      currency = "USD"
      amount = 100.0

  submit accountant do
    createCmd CashBalance with
      accountant
      owner
      cash
      location = InHand

  submit accountant do
    createCmd CashBalance with
      accountant
      owner
      cash
      location = InAccount account
```

The way to read the declaration of `Location` is *A Location either has value* `InHand` *OR has a value* `InAccount a` *where* `a` *is of type Account* . This is quite an explicit way to say that there may or may not be an `Account` associated with a `CashBalance` and gives both cases suggestive names.

Another option is to use the built-in `Optional` type. The `None` value of type `Optional a` is the closest Daml has to a `null` value:

```
data Optional a
  = None
  | Some a
    deriving (Eq, Show)
```

Variant types where none of the data constructors take a parameter are called enums:

```
data DayOfWeek
  = Monday
  | Tuesday
  | Wednesday
  | Thursday
  | Friday
  | Saturday
  | Sunday
    deriving (Eq, Show)
```

To access the data in variants, you need to distinguish the different possible cases. For example, you can no longer access the account number of a `Location` directly, because if it is `InHand`, there may be no account number.

To do this, you can use *pattern matching* and either throw errors or return compatible types for all cases:

```
{-
-- Commented out as `Either` is defined in the standard library.
data Either a b
  = Left a
  | Right b
-}

variant_access_test = script do
  let
    l : Either Int Text = Left 1
    r : Either Int Text = Right "r"

    -- If we know that `l` is a `Left`, we can error on the `Right` case.
    l_value = case l of
      Left i -> i
      Right i -> error "Expecting Left"
    -- Comment out at your own peril
    {-
    r_value = case r of
      Left i -> i
      Right i -> error "Expecting Left"
    -}

    -- If we are unsure, we can return an `Optional` in both cases
    ol_value = case l of
      Left i -> Some i
      Right i -> None
    or_value = case r of
      Left i -> Some i
      Right i -> None

    -- If we don't care about values or even constructors, we can use wildcards
```

(continues on next page)

(continued from previous page)

```
    l_value2 = case l of
      Left i -> i
      Right _ -> error "Expecting Left"
    l_value3 = case l of
      Left i -> i
      _ -> error "Expecting Left"

    day = Sunday
    weekend = case day of
      Saturday -> True
      Sunday -> True
      _ -> False

  assert (l_value == 1)
  assert (l_value2 == 1)
  assert (l_value3 == 1)
  assert (ol_value == Some 1)
  assert (or_value == None)
  assert weekend
```

1.11.4.3 Manipulate Data

You've got all the ingredients to build rich types expressing the data you want to be able to write to the ledger, and you have seen how to create new values and read fields from values. But how do you manipulate values once created?

All data in Daml is immutable, meaning once a value is created, it will never change. Rather than changing values, you create new values based on old ones with some changes applied:

```
manipulation_demo = script do
  let
    eq_record = EqRecord with
      my_txt = "Text"
      my_int = 2
      my_dec = 2.5
      my_list = ["One", "Two", "Three"]

    -- A verbose way to change `eq_record`
    changed_record = EqRecord with
      my_txt = eq_record.my_txt
      my_int = 3
      my_dec = eq_record.my_dec
      my_list = eq_record.my_list

    -- A better way
    better_changed_record = eq_record with
      my_int = 3

    record_with_changed_list = eq_record with
      my_list = "Zero" :: eq_record.my_list

  assert (eq_record.my_int == 2)
  assert (changed_record == better_changed_record)
```

(continues on next page)

(continued from previous page)

```
  -- The list on `eq_record` can't be changed.
  assert (eq_record.my_list == ["One", "Two", "Three"])
  -- The list on `record_with_changed_list` is a new one.
  assert (record_with_changed_list.my_list == ["Zero", "One", "Two", "Three"])
```

`changed_record` and `better_changed_record` are each a copy of `eq_record` with the field `my_int` changed. `better_changed_record` shows the recommended way to change fields on a record. The syntax is almost the same as for a new record, but the record name is replaced with the old value: `eq_record with` instead of `EqRecord with`. The `with` block no longer needs to give values to all fields of `EqRecord`. Any missing fields are taken from `eq_record`.

Throughout the script, `eq_record` never changes. The expression `"Zero" :: eq_record.my_list` doesn't change the list in-place, but creates a new list, which is `eq_record.my_list` with an extra element in the beginning.

1.11.4.4 Contract Keys

Daml's type system lets you store richly structured data on Daml templates, but just like most database schemas have more than one table, Daml contract models often have multiple templates that reference each other. For example, you may not want to store your bank and account information on each individual cash balance contract, but instead store those on separate contracts.

You have already met the type `ContractId a`, which references a contract of type `a`. The below shows a contract model where `Account` is split out into a separate template and referenced by `ContractId`, but it also highlights a big problem with that kind of reference: just like data, contracts are immutable. They can only be created and archived, so if you want to change the data on a contract, you end up archiving the original contract and creating a new one with the changed data. That makes contract IDs very unstable, and can cause stale references.

```
data Bank = Bank with
  party : Party
  address: Text
  telephone : Text
    deriving (Eq, Show)

template Account
  with
    accountant : Party
    owner : Party
    number : Text
    bank : Bank
  where
    signatory accountant

data Cash = Cash with
  currency : Text
  amount : Decimal
    deriving (Eq, Show)

template CashBalance
  with
    accountant : Party
    cash : Cash
```

(continues on next page)

(continued from previous page)

```
    account : ContractId Account
  where
    signatory accountant

id_ref_test = do
  accountant <- allocateParty "Bob"
  owner <- allocateParty "Alice"
  bank_party <- allocateParty "Bank"
  let
    bank = Bank with
      party = bank_party
      address = "High Street"
      telephone = "012 3456 789"
    cash = Cash with
      currency = "USD"
      amount = 100.0

  accountCid <- submit accountant do
     createCmd Account with
      accountant
      owner
      bank
      number = "ABC123"

  balanceCid <- submit accountant do
    createCmd CashBalance with
      accountant
      cash
      account = accountCid

  -- Now the accountant updates the telephone number for the bank on the account
  Some account <- queryContractId accountant accountCid
  new_accountCid <- submit accountant do
    archiveCmd accountCid
    cid <- createCmd account with
      bank = account.bank with
        telephone = "098 7654 321"
    pure cid

  -- The `account` field on the balance now refers to the archived
  -- contract, so this will fail.
  Some balance <- queryContractId accountant balanceCid
  optAccount <- queryContractId accountant balance.account
  optAccount === None
```

The script above uses the `queryContractId` function, which retrieves the arguments of an active contract using its contract ID. If there is no active contract with the given identifier visible to the given party, `queryContractId` returns `None`. Here, we use a pattern match on `Some` which will abort the script if `queryContractId` returns `None`.

Note that, for the first time, the party submitting a transaction is doing more than one thing as part of that transaction. To create `new_account`, the accountant archives the old account and creates a new account, all in one transaction. More on building transactions in *Composing Choices*.

You can define *stable* keys for contracts using the `key` and `maintainer` keywords. `key` defines the primary key of a template, with the ability to look up contracts by key, and a uniqueness constraint

in the sense that only one contract of a given template and with a given key value can be active at a time:

```
data Bank = Bank with
  party : Party
  address: Text
  telephone : Text
    deriving (Eq, Show)

data AccountKey = AccountKey with
  accountant : Party
  number : Text
  bank_party : Party
    deriving (Eq, Show)

template Account
  with
    accountant : Party
    owner : Party
    number : Text
    bank : Bank
  where
    signatory accountant

    key AccountKey with
        accountant
        number
        bank_party = bank.party
      : AccountKey
    maintainer key.accountant

data Cash = Cash with
  currency : Text
  amount : Decimal
    deriving (Eq, Show)

template CashBalance
  with
    accountant : Party
    cash : Cash
    account : AccountKey
  where
    signatory accountant

id_ref_test = do
  accountant <- allocateParty "Bob"
  owner <- allocateParty "Alice"
  bank_party <- allocateParty "Bank"
  let
    bank = Bank with
      party = bank_party
      address = "High Street"
      telephone = "012 3456 789"
    cash = Cash with
      currency = "USD"
      amount = 100.0
```

(continues on next page)

(continued from previous page)

```
  accountCid <- submit accountant do
     createCmd Account with
      accountant
      owner
      bank
      number = "ABC123"

  Some account <- queryContractId accountant accountCid
  balanceCid <- submit accountant do
    createCmd CashBalance with
      accountant
      cash
      account = key account

  -- Now the accountant updates the telephone number for the bank on the account
  Some account <- queryContractId accountant accountCid
  new_accountCid <- submit accountant do
    archiveCmd accountCid
    cid <- createCmd account with
      bank = account.bank with
        telephone = "098 7654 321"
    pure cid

  -- Thanks to contract keys, the current account contract is fetched
  Some balance <- queryContractId accountant balanceCid
  Some (new_cid, new_account : Account) <- queryContractKey accountant balance.
↪account
  new_cid === new_accountCid
  new_account =/= account
```

Since Daml is designed to run on distributed systems, you have to assume that there is no global entity that can guarantee uniqueness, which is why each `key` expression must come with a `maintainer` expression. `maintainer` takes one or several parties, all of which have to be signatories of the contract and be part of the key. That way the index can be partitioned amongst sets of maintainers, and each set of maintainers can independently ensure the uniqueness constraint on their piece of the index. The constraint that maintainers are part of the key is ensured by only having the variable *key* in each maintainer expression.

Instead of calling `queryContractId` to get the contract arguments associated with a given contract identifier, use `queryContractKey`. `queryContractKey` takes a value of type `AccountKey` and returns an optional tuple. In this case, that optional tuple is of type `Optional (ContractId Account, Account)`. After archiving the old account (to change the phone number), you can still fetch the account using the existing, unmodified `balance`. Where the `ContractId Account` is different for the new account, the `AccountKey` is the same.

When calling `queryContractKey` a single key type could be used as the key for multiple templates. Consequently, you need to tell the compiler what type of contract the key is referencing. You can do that with a type annotation on the returned value.

1.11.4.5 Next Up

You can now define data schemas for the ledger, read, write and delete data from the ledger, and use keys to reference and look up data in a stable fashion.

In *Transform Data Using Choices* you'll learn how to define data transformations and give other parties the right to manipulate data in restricted ways.

1.11.5 Transform Data Using Choices

In the example in *Contract Keys* the accountant party wanted to change some data on a contract. They did so by archiving the contract and re-creating it with the updated data. That works because the accountant is the sole signatory on the `Account` contract defined there.

But what if the accountant wanted to allow the bank to change their own telephone number? Or what if the owner of a `CashBalance` should be able to transfer ownership to someone else?

In this section you will learn about how to define simple data transformations using *choices* and how to delegate the right to *exercise* these choices to other parties.

Hint: Remember that you can load all the code for this section into a folder called `intro4` by running `daml new intro4 --template daml-intro-4`

1.11.5.1 Choices as Methods

If you think of templates as classes and contracts as objects, where are the methods?

Take as an example a `Contact` contract on which the contact owner wants to be able to change the telephone number, just like on the `Account` in *Contract Keys*. Rather than requiring them to manually look up the contract, archive the old one and create a new one, you can provide them a convenience method on `Contact`:

```
template Contact
  with
    owner : Party
    party : Party
    address : Text
    telephone : Text
  where
    signatory owner
    observer party

    choice UpdateTelephone
      : ContractId Contact
      with
        newTelephone : Text
      controller owner
      do
        create this with
          telephone = newTelephone
```

The above defines a *choice* called `UpdateTelephone`. Choices are part of a contract template. They're permissioned functions that result in an `Update`. Using choices, authority can be passed around, allowing the construction of complex transactions.

Let's unpack the code snippet above:

> The first line, `choice UpdateTelephone` indicates a choice definition, `UpdateTelephone` is the name of the choice. It starts a new block in which that choice is defined.
> `: ContractId Contact` is the return type of the choice.
> This particular choice archives the current `Contact`, and creates a new one. What it returns is a reference to the new contract, in the form of a `ContractId Contact`
> The following `with` block is that of a record. Just like with templates, in the background, a new record type is declared: `data UpdateTelephone = UpdateTelephone with`
> The line `controller owner` says that this choice is *controlled* by `owner`, meaning `owner` is the only party that is allowed to *exercise* them.
> The `do` starts a block defining the action the choice should perform when exercised. In this case a new `Contact` is created.
> The new `Contact` is created using `this with`. `this` is a special value available within the `where` block of templates and takes the value of the current contract's arguments.

There is nothing here explicitly saying that the current `Contact` should be archived. That's because choices are *consuming* by default. That means when the above choice is exercised on a contract, that contract is archived.

As mentioned in *Data Types*, within a choice we use `create` instead of `createCmd`. Whereas `createCmd` builds up a list of commands to be sent to the ledger, `create` builds up a more flexible `Update` that is executed directly by the ledger. You might have noticed that `create` returns an `Update (ContractId Contact)`, not a `ContractId Contact`. As a `do` block always returns the value of the last statement within it, the whole `do` block returns an `Update`, but the return type on the choice is just a `ContractId Contact`. This is a convenience. Choices *always* return an `Update` so for readability it's omitted on the type declaration of a choice.

Now to exercise the new choice in a script:

```
choice_test = do
  owner <- allocateParty "Alice"
  party <- allocateParty "Bob"

  contactCid <- submit owner do
     createCmd Contact with
      owner
      party
      address = "1 Bobstreet"
      telephone = "012 345 6789"

  -- Bob can't change his own telephone number as Alice controls
  -- that choice.
  submitMustFail party do
    exerciseCmd contactCid UpdateTelephone with
      newTelephone = "098 7654 321"

  newContactCid <- submit owner do
    exerciseCmd contactCid UpdateTelephone with
      newTelephone = "098 7654 321"

  Some newContact <- queryContractId owner newContactCid
```

(continues on next page)

(continued from previous page)

```
assert (newContact.telephone == "098 7654 321")
```

You exercise choices using the `exercise` function, which takes a `ContractId a`, and a value of type `c`, where `c` is a choice on template `a`. Since `c` is just a record, you can also just fill in the choice parameters using the `with` syntax you are already familiar with.

`exerciseCmd` returns a `Commands r` where `r` is the return type specified on the choice, allowing the new `ContractId Contact` to be stored in the variable `newContactCid`. Just like for `createCmd` and `create`, there is also `exerciseCmd` and `exercise`. The versions with the `cmd` suffix is always used on the client side to build up the list of commands on the ledger. The versions without the suffix are used within choices and are executed directly on the server.

There is also `createAndExerciseCmd` and `createAndExercise` which we have seen in the previous section. This allows you to create a new contract with the given arguments and immediately exercise a choice on it. For a consuming choice, this archives the contract so the contract is created and archived within the same transaction.

1.11.5.2 Choices as Delegation

Up to this point all the contracts only involved one party. `party` may have been stored as `Party` field in the above, which suggests they are actors on the ledger, but they couldn't see the contracts, nor change them in any way. It would be reasonable for the party for which a `Contact` is stored to be able to update their own address and telephone number. In other words, the `owner` of a `Contact` should be able to *delegate* the right to perform a certain kind of data transformation to `party`.

The below demonstrates this using an `UpdateAddress` choice and corresponding extension of the script:

```
    choice UpdateAddress
      : ContractId Contact
      with
        newAddress : Text
      controller party
      do
        create this with
          address = newAddress
```

```
  newContactCid <- submit party do
    exerciseCmd newContactCid UpdateAddress with
      newAddress = "1-10 Bobstreet"

  Some newContact <- queryContractId owner newContactCid

  assert (newContact.address == "1-10 Bobstreet")
```

If you open the script view in the IDE, you will notice that Bob sees the `Contact`. This is because `party` is specified as an `observer` in the template, and in this case Bob is the `party`. More on *observers* later, but in short, they get to see any changes to the contract.

1.11.5.3 Choices In the Ledger Model

In *Basic Contracts* you learned about the high-level structure of a Daml ledger. With choices and the *exercise* function, you have the next important ingredient to understand the structure of the ledger and transactions.

A *transaction* is a list of *actions*, and there are just four kinds of action: `create`, `exercise`, `fetch` and `key assertion`.

> A `create` action creates a new contract with the given arguments and sets its status to *active*.
> A `fetch` action checks the existence and activeness of a contract.
> An `exercise` action exercises a choice on a contract resulting in a transaction (list of sub-actions) called the *consequences*. Exercises come in two kinds called `consuming` and `non-consuming`. `consuming` is the default kind and changes the contract's status from *active* to *archived*.
> A `key assertion` records the assertion that the given contract key (see *Contract Keys*) is not assigned to any active contract on the ledger.

Each action can be visualized as a tree, where the action is the root node, and its children are its consequences. Every consequence may have further consequences. As `fetch`, `create` and `key assertion` actions have no consequences, they are always leaf nodes. You can see the actions and their consequences in the transaction view of the above script:

```
Transactions:
  TX 0 1970-01-01T00:00:00Z (Contact:46:17)
  #0:0
  |   consumed by: #2:0
  |   referenced by #2:0
  |   disclosed to (since): 'Alice' (0), 'Bob' (0)
  └─> 'Alice' creates Contact:Contact
              with
                owner = 'Alice'; party = 'Bob'; address = "1 Bobstreet";
↪telephone = "012 345 6789"

  TX 1 1970-01-01T00:00:00Z
    mustFailAt actAs: {'Bob'} readAs: {} (Contact:55:3)

  TX 2 1970-01-01T00:00:00Z (Contact:59:20)
  #2:0
  |   disclosed to (since): 'Alice' (2), 'Bob' (2)
  └─> 'Alice' exercises UpdateTelephone on #0:0 (Contact:Contact)
              with
                newTelephone = "098 7654 321"
      children:
      #2:1
      |   consumed by: #3:0
      |   referenced by #3:0
      |   disclosed to (since): 'Alice' (2), 'Bob' (2)
      └─> 'Alice' creates Contact:Contact
                  with
                    owner = 'Alice'; party = 'Bob'; address = "1 Bobstreet";
↪telephone = "098 7654 321"

  TX 3 1970-01-01T00:00:00Z (Contact:69:20)
  #3:0
  |   disclosed to (since): 'Alice' (3), 'Bob' (3)
```

(continues on next page)

(continued from previous page)

```
  └─> 'Bob' exercises UpdateAddress on #2:1 (Contact:Contact)
            with
              newAddress = "1-10 Bobstreet"
      children:
      #3:1
      |   disclosed to (since): 'Alice' (3), 'Bob' (3)
      └─> 'Alice' creates Contact:Contact
                  with
                    owner = 'Alice';
                    party = 'Bob';
                    address = "1-10 Bobstreet";
                    telephone = "098 7654 321"

Active contracts:  #3:1

Return value: {}
```

There are four commits corresponding to the four `submit` statements in the script. Within each commit, we see that it's actually actions that have IDs of the form `#commit_number:action_number`. Contract IDs are just the ID of their `create` action.

So commits `#2` and `#3` contain `exercise` actions with IDs `#2:0` and `#3:0`. The `create` actions of the updated `Contact` contracts, `#2:1` and `#3:1`, are indented and found below a line reading `children:`, making the tree structure apparent.

The Archive Choice

You may have noticed that there is no archive action. That's because `archive cid` is just shorthand for `exercise cid Archive`, where `Archive` is a choice implicitly added to every template, with the signatories as controllers.

1.11.5.4 A Simple Cash Model

With the power of choices, you can build your first interesting model: issuance of cash IOUs (I owe you). The model presented here is simpler than the one in *Data Types* as it's not concerned with the location of the physical cash, but merely with liabilities:

```
-- Copyright (c) 2023 Digital Asset (Switzerland) GmbH and/or its affiliates. All
↪rights reserved.
-- SPDX-License-Identifier: Apache-2.0


module SimpleIou where

import Daml.Script

data Cash = Cash with
  currency : Text
  amount : Decimal
    deriving (Eq, Show)

template SimpleIou
```

(continues on next page)

(continued from previous page)

```
  with
    issuer : Party
    owner : Party
    cash : Cash
  where
    signatory issuer
    observer owner

    choice Transfer
      : ContractId SimpleIou
      with
        newOwner : Party
      controller owner
      do
        create this with owner = newOwner

test_iou = script do
  alice <- allocateParty "Alice"
  bob <- allocateParty "Bob"
  charlie <- allocateParty "Charlie"
  dora <- allocateParty "Dora"

  -- Dora issues an Iou for $100 to Alice.
  iou <- submit dora do
    createCmd SimpleIou with
      issuer = dora
      owner = alice
      cash = Cash with
        amount = 100.0
        currency = "USD"

  -- Alice transfers it to Bob.
  iou2 <- submit alice do
    exerciseCmd iou Transfer with
      newOwner = bob

  -- Bob transfers it to Charlie.
  submit bob do
    exerciseCmd iou2 Transfer with
      newOwner = charlie
```

The above model is fine as long as everyone trusts Dora. Dora could revoke the *SimpleIou* at any point by archiving it. However, the provenance of all transactions would be on the ledger so the owner could *prove* that Dora was dishonest and cancelled her debt.

1.11.5.5 Next Up

You can now store and transform data on the ledger, even giving other parties specific write access through choices.

In *Add Constraints to a Contract*, you will learn how to restrict data and transformations further. In that context, you will also learn about time on Daml ledgers, `do` blocks and `<-` notation within those.

1.11.6 Add Constraints to a Contract

You will often want to constrain the data stored or the allowed data transformations in your contract models. In this section, you will learn about the two main mechanisms provided in Daml:

The `ensure` keyword.
The `assert`, `abort` and `error` keywords.

To make sense of the latter, you'll also learn more about the `Update` and `Script` types and `do` blocks, which will be good preparation for *Composing Choices*, where you will use `do` blocks to compose choices into complex transactions.

Lastly, you will learn about time on the ledger and in Daml Script.

Hint: Remember that you can load all the code for this section into a folder called `intro5` by running `daml new intro5 --template daml-intro-5`

1.11.6.1 Template Preconditions

The first kind of restriction you may want to put on the contract model are called *template pre-conditions*. These are simply restrictions on the data that can be stored on a contract from that template.

Suppose, for example, that the `SimpleIou` contract from *A Simple Cash Model* should only be able to store positive amounts. You can enforce this using the `ensure` keyword:

```
template SimpleIou
  with
    issuer : Party
    owner : Party
    cash : Cash
  where
    signatory issuer
    observer owner

    ensure cash.amount > 0.0
```

The `ensure` keyword takes a single expression of type `Bool`. If you want to add more restrictions, use logical operators `&&`, `||` and `not` to build up expressions. The below shows the additional restriction that currencies are three capital letters:

```
        && T.length cash.currency == 3
        && T.isUpper cash.currency
```

Hint: The `T` here stands for the `DA.Text` standard library which has been imported using `import DA.Text as T`:

```
test_restrictions = do
  alice <- allocateParty "Alice"
  bob <- allocateParty "Bob"
  dora <- allocateParty "Dora"

  -- Dora can't issue negative Ious.
  submitMustFail dora do
    createCmd SimpleIou with
      issuer = dora
      owner = alice
      cash = Cash with
        amount = -100.0
        currency = "USD"

  -- Or even zero Ious.
  submitMustFail dora do
    createCmd SimpleIou with
      issuer = dora
      owner = alice
      cash = Cash with
        amount = 0.0
        currency = "USD"

  -- Nor positive Ious with invalid currencies.
  submitMustFail dora do
    createCmd SimpleIou with
      issuer = dora
      owner = alice
      cash = Cash with
        amount = 100.0
        currency = "Swiss Francs"

  -- But positive Ious still work, of course.
  iou <- submit dora do
    createCmd SimpleIou with
      issuer = dora
      owner = alice
      cash = Cash with
        amount = 100.0
        currency = "USD"
```

1.11.6.2 Assertions

A second common kind of restriction is one on data transformations.

For example, the simple Iou in *A Simple Cash Model* allowed the no-op where the `owner` transfers to themselves. You can prevent that using an `assert` statement, which you have already encountered in the context of scripts.

`assert` does not return an informative error so often it's better to use the function `assertMsg`, which takes a custom error message:

```
    choice Transfer
      : ContractId SimpleIou
      with
        newOwner : Party
      controller owner
      do
        assertMsg "newOwner cannot be equal to owner." (owner /= newOwner)
        create this with owner = newOwner
```

```
  -- Alice can't transfer to herself...
  submitMustFail alice do
    exerciseCmd iou Transfer with
      newOwner = alice

  -- ... but can transfer to Bob.
  iou2 <- submit alice do
    exerciseCmd iou Transfer with
      newOwner = bob
```

Similarly, you can write a `Redeem` choice, which allows the `owner` to redeem an `Iou` *during business hours on weekdays*. The `Redeem` choice implementation below confirms that `getTime` returns a value that is during business hours on weekdays. If all those checks pass, the choice does not do anything other than archive the `SimpleIou`. (This assumes that actual cash changes hands off-ledger:)

```
    choice Redeem
      : ()
      controller owner
      do
        now <- getTime
        let
          today = toDateUTC now
          dow = dayOfWeek today
          timeofday = now `subTime` time today 0 0 0
          hrs = convertRelTimeToMicroseconds timeofday / 3600000000
        if (hrs < 8 || hrs > 18) then
          abort $ "Cannot redeem outside business hours. Current time: " <> show
↪timeofday
        else case dow of
          Saturday -> abort "Cannot redeem on a Saturday."
          Sunday -> abort "Cannot redeem on a Sunday."
          _ -> return ()
```

In the above example, the time is taken apart into day of week and hour of day using standard library functions from `DA.Date` and `DA.Time`. The hour of the day is checked to be in the range from 8 to 18. The day of week is checked to not be Saturday or Sunday.

The following example shows how the `Redeem` choice is exercised in a script:

```
-- June 1st 2019 is a Saturday.
setTime (time (date 2019 Jun 1) 0 0 0)
-- Bob cannot redeem on a Saturday.
submitMustFail bob do
  exerciseCmd iou2 Redeem

-- Not even at mid-day.
passTime (hours 12)
-- Bob cannot redeem on a Saturday.
submitMustFail bob do
  exerciseCmd iou2 Redeem

-- Bob also cannot redeem at 6am on a Monday.
passTime (hours 42)
submitMustFail bob do
  exerciseCmd iou2 Redeem

-- Bob can redeem at 8am on Monday.
passTime (hours 2)
submit bob do
  exerciseCmd iou2 Redeem
```

For the purposes of testing the `Redeem` choice, the above code sets and advances the ledger time with the `setTime` and `passTime` functions respectively. Exercising the choice should fail or should not fail depending on the day of week and the time of day. While that is straightforward, the issue of time on a Daml ledger is worthy of more discussion.

1.11.6.3 Time on Daml Ledgers

Each transaction on a Daml ledger has two timestamps: the *ledger time (LT)* and the *record time (RT)*.

Ledger time (LT) is the time associated with a transaction in the ledger model, as determined by the participant. It is the time of a transaction from a business and application perspective. When you call `getTime`, it is the LT that is returned. The LT is used when reasoning about related transactions and commits. The LT can be compared with other LTs to guarantee model consistency. For example, LTs are used to enforce that no transaction depends on a contract that does not exist. This is the requirement known as causal monotonicity.

Record time (RT) is the time assigned by the persistence layer. It represents the time that the transaction is physically recorded. For example, The backing database ledger has assigned the timestamp of such-and-such time to this transaction. The only purpose of the RT is to ensure that transactions are being recorded in a timely manner.

Each Daml ledger has a policy on the allowed difference between LT and RT called the *skew*. A consistent zero-skew is not feasible because this is a distributed system. If it is too far off, the transaction will be rejected. This is the requirement known as bounded skew. The RT is not relevant beyond this determination of skew.

Returning to the theme of *business hours*, consider the following example: Suppose that the ledger had a skew of 10 seconds. At 17:59:55, just before the end of business hours, Alice submits a transaction to redeem an Iou. One second later, the transaction is assigned an LT of 17:59:56. However, there still may be a few seconds before the transaction is persisted to the underlying storage. For example, the transaction might be written in the underlying backing store at 18:00:06, *after business hours*.

Because LT is within business hours and LT - RT <= 10 seconds, the transaction will not be rejected.

For details, see *Background concepts - time*.

Time in Test Scripts

For tests, you can set time using the following functions:

`setTime`, which sets the ledger time to the given time.
`passTime`, which takes a `RelTime` (a relative time) and moves the ledger by that much.

On a distributed Daml ledger, there are no guarantees that LT or RT are strictly increasing. The only guarantee is that ledger time is increasing *with causality*. That is, if a transaction `TX2` depends on a transaction `TX1`, then the ledger enforces that the LT of `TX2` is greater than or equal to that of `TX1`.

The following script illustrates that idea by moving the logical time back by three days and then trying to exercise a choice on a contract *that hasn't been created yet*. That fails, as you would hope.

```
iou3 <- submit dora do
  createCmd SimpleIou with
    issuer = dora
    owner = alice
    cash = Cash with
      amount = 100.0
      currency = "USD"

passTime (days (-3))
submitMustFail alice do
  exerciseCmd iou3 Redeem
```

1.11.6.4 Actions and `do` Blocks

You have come across `do` blocks and `<-` notations in two contexts by now: `Script` and `Update`. Both of these are examples of an `Action`, also called a *Monad* in functional programming. You can construct `Actions` conveniently using `do` notation.

Understanding `Actions` and `do` blocks is therefore crucial to being able to construct correct contract models and test them, so this section will explain them in some detail.

Pure Expressions Compared to Actions

Expressions in Daml are pure in the sense that they have no side-effects: they neither read nor modify any external state. If you know the value of all variables in scope and write an expression, you can work out the value of that expression on pen and paper.

However, the expressions you've seen that used the `<-` notation are not like that. For example, take `getTime`, which is an `Action`. Here's the example we used earlier:

```
now <- getTime
```

You cannot work out the value of `now` based on any variable in scope. To put it another way, there is no expression `expr` that you could put on the right hand side of `now = expr`. To get the ledger time, you must be in the context of a submitted transaction, and then look at that context.

Similarly, you've come across `fetch`. If you have `cid : ContractId Account` in scope and you come across the expression `fetch cid`, you can't evaluate that to an `Account` so you can't write `account = fetch cid`. To do so, you'd have to have a ledger you can look that contract ID up on.

Actions and Impurity

Actions are a way to handle such impure expressions. `Action a` is a type class with a single parameter `a`, and `Update` and `Script` are instances of `Action`. A value of such a type `m a` where `m` is an instance of `Action` can be interpreted as a recipe for an action of type `m`, which, when executed, returns a value `a`.

You can always write a recipe using just pen and paper, but you can't cook it up unless you are in the context of a kitchen with the right ingredients and utensils. When cooking the recipe you have an effect – you change the state of the kitchen – and a return value – the thing you leave the kitchen with.

> An `Update a` is a recipe to update a Daml ledger, which, when committed, has the effect of changing the ledger, and returns a value of type `a`. An update to a Daml ledger is a transaction so equivalently, an `Update a` is a recipe to construct a transaction, which, when executed in the context of a ledger, returns a value of type `a`.
> A `Script a` is a recipe for a test, which, when performed against a ledger, has the effect of changing the ledger in ways analogous to those available via the API, and returns a value of type `a`.

Expressions like `getTime`, `allocateParty party`, `passTime time`, `submit party commands`, `create contract` and `exercise choice` should make more sense in that light. For example:

> `getTime : Update Time` is the recipe for an empty transaction that also happens to return a value of type `Time`.
> `passTime (days 10) : Script ()` is a recipe for a transaction that doesn't submit any transactions, but has the side-effect of changing the LT of the test ledger. It returns `()`, also called `Unit` and can be thought of as a zero-tuple.
> `create iou : Update (ContractId Iou)`, where `iou : Iou` is a recipe for a transaction consisting of a single `create` action, and returns the contract id of the created contract if successful.
> `submit alice (createCmd iou) : Script (ContractId Iou)` is a recipe for a script in which Alice sends the command `createCmd iou` to the ledger which produces a transaction and a return value of type `ContractId Iou` and returns that back to Alice.

`Commands` is a bit more restricted than `Script` and `Update` as it represents a list of independent commands sent to the ledger. You can still use `do` blocks but if you have more than one command in a single `do` block you need to enable the `ApplicativeDo` extension at the beginning of your file. In addition to that, the last statement in such a `do` block must be of the form `return expr` or `pure expr`. `Applicative` is a more restricted version of `Action` that enforces that there are no dependencies between commands. If you do have dependencies between commands, you can always wrap it in a choice in a helper template and call that via `createAndExerciseCmd` just like we did to call `fetchByKey`. Alternatively, if you do not need them to be part of the same transaction, you can make multiple calls to `submit`:

```
{-# LANGUAGE ApplicativeDo #-}
module Restrictions where
```

Chain Actions With `do` Blocks

An action followed by another action, possibly depending on the result of the first action, is just another action. Specifically:

> A transaction is a list of actions. So a transaction followed by another transaction is again a transaction.
> A script is a list of interactions with the ledger (`submit`, `allocateParty`, `passTime`, etc). So a script followed by another script is again a script.

This is where `do` blocks come in. `do` blocks allow you to build complex actions from simple ones, using the results of earlier actions in later ones:

```
sub_script1 (alice, dora) = do
  submit dora do
    createCmd SimpleIou with
      issuer = dora
      owner = alice
      cash = Cash with
        amount = 100.0
        currency = "USD"

sub_script2 = do
  passTime (days 1)
  passTime (days (-1))
  return 42

sub_script3 (bob, dora) = do
  submit dora do
    createCmd SimpleIou with
      issuer = dora
      owner = bob
      cash = Cash with
        amount = 100.0
        currency = "USD"

main_: Script () = do
  dora <- allocateParty "Dora"
  alice <- allocateParty "Alice"
  bob <- allocateParty "Bob"

  iou1 <- sub_script1 (alice, dora)
  sub_script2
  iou2 <- sub_script3 (bob, dora)

  submit dora do
    archiveCmd iou1
    archiveCmd iou2
    pure ()
```

Above, we see `do` blocks in action for both `Script` and `Update`.

Wrap Values in Actions

You may already have noticed the use of `return` in the redeem choice. `return x` is a no-op action which returns value `x` so `return 42 : Update Int`. Since `do` blocks always return the value of their last action, `sub_script2 : Script Int`.

1.11.6.5 Failing Actions

Not only are `Update` and `Script` examples of `Action`, they are both examples of actions that can fail, e.g. because a transaction is illegal or the party retrieved via `allocateParty` doesn't exist on the ledger.

Each has a special action `abort txt` that represents failure, and that takes on type `Update ()` or `Script ()` depending on context .

Transactions succeed or fail *atomically* as a whole. Scripts on the other hand do not fail atomically: while each `submit` is atomic, if a `submit` succeeded and the script fails later, the effects of that `submit` will still be applied to the ledger.

The last expression in the `do` block of the `Redeem` choice is a pattern matching expression on `dow`. It has type `Update ()` and is either an `abort` or `return` depending on the day of week. So during the week, it's a no-op and on weekends, it's the special failure action. Thanks to the atomicity of transactions, no transaction can ever make use of the `Redeem` choice on weekends, because it fails the entire transaction.

1.11.6.6 A Sample Action

If the above didn't make complete sense, here's another example to explain what actions are more generally, by creating a new type that is also an action. `CoinGame a` is an `Action a` in which a `Coin` is flipped. The `Coin` is a pseudo-random number generator and each flip has the effect of changing the random number generator's state. Based on the `Heads` and `Tails` results, a return value of type `a` is calculated:

```
data Face = Heads | Tails
  deriving (Eq, Show, Enum)

data CoinGame a = CoinGame with
  play : Coin -> (Coin, a)

flipCoin : CoinGame Face
getCoin : Script Coin
```

A `CoinGame a` exposes a function `play` which takes a `Coin` and returns a new `Coin` and a result `a`. More on the `->` syntax for functions later.

`Coin` and `play` are deliberately left obscure in the above. All you have is an action `getCoin` to get your hands on a `Coin` in a `Script` context and an action `flipCoin` which represents the simplest possible game: a single coin flip resulting in a `Face`.

You can't play any `CoinGame` game on pen and paper as you don't have a coin, but you can write down a script or recipe for a game:

```
coin_test = do
  -- The coin is pseudo-random on LT so change the parameter to change the game.
  setTime (time (date 2019 Jun 1) 0 0 0)
  passTime (seconds 2)
  coin <- getCoin
  let
    game = do
      f1r <- flipCoin
      f2r <- flipCoin
      f3r <- flipCoin

      if all (== Heads) [f1r, f2r, f3r]
        then return "Win"
        else return "Loss"
    (newCoin, result) = game.play coin

  assert (result == "Win")
```

The `game` expression is a `CoinGame` in which a coin is flipped three times. If all three tosses return `Heads`, the result is `"Win"`, or else `"Loss"`.

In a `Script` context you can get a `Coin` using the `getCoin` action, which uses the LT to calculate a seed, and play the game.

Somehow the `Coin` is threaded through the various actions. If you want to look through the looking glass and understand in-depth what's going on, you can look at the source file to see how the `CoinGame` action is implemented, though be warned that the implementation uses a lot of Daml features we haven't introduced yet in this introduction.

More generally, if you want to learn more about Actions (aka Monads), we recommend a general course on functional programming, and Haskell in particular. See *The Haskell Connection* for some suggestions.

1.11.6.7 Errors

Above, you've learnt about `assertMsg` and `abort`, which represent (potentially) failing actions. Actions only have an effect when they are performed, so the following script succeeds or fails depending on the value of `abortScript`:

```
nonPerformedAbort = do
  let abortScript = False
  let failingAction : Script () = abort "Foo"
  let successfulAction : Script () = return ()
  if abortScript then failingAction else successfulAction
```

However, what about errors in contexts other than actions? Suppose we wanted to implement a function `pow` that takes an integer to the power of another positive integer. How do we handle that the second parameter has to be positive?

One option is to make the function explicitly partial by returning an `Optional`:

```
optPow : Int -> Int -> Optional Int
optPow base exponent
 | exponent == 0 = Some 1
 | exponent > 0 =
```

(continues on next page)

(continued from previous page)

```
  let Some result = optPow base (exponent - 1)
  in Some (base * result)
 | otherwise = None
```

This is a useful pattern if we need to be able to handle the error case, but it also forces us to always handle it as we need to extract the result from an `Optional`. We can see the impact on convenience in the definition of the above function. In cases, like division by zero or the above function, it can therefore be preferable to fail catastrophically instead:

```
errPow : Int -> Int -> Int
errPow base exponent
 | exponent == 0 = 1
 | exponent > 0 = base * errPow base (exponent - 1)
 | otherwise = error "Negative exponent not supported"
```

The big downside to this is that even unused errors cause failures. The following script will fail, because `failingComputation` is evaluated:

```
nonPerformedError = script do
  let causeError = False
  let failingComputation = errPow 1 (-1)
  let successfulComputation = errPow 1 1
  return if causeError then failingComputation else successfulComputation
```

`error` should therefore only be used in cases where the error case is unlikely to be encountered, and where explicit partiality would unduly impact usability of the function.

1.11.6.8 Next Up

You can now specify a precise data and data-transformation model for Daml ledgers. In *Parties and Authority*, you will learn how to properly involve multiple parties in contracts, how authority works in Daml, and how to build contract models with strong guarantees in contexts with mutually distrusting entities.

1.11.7 Parties and Authority

Daml is designed for distributed applications involving mutually distrusting parties. In a well-constructed contract model, all parties have strong guarantees that nobody cheats or circumvents the rules laid out by templates and choices.

In this section you will learn about Daml's authorization rules and how to develop contract models that give all parties the required guarantees. In particular, you'll learn how to:

Pass authority from one contract to another
Write advanced choices
Reason through Daml's Authorization model

Hint: Remember that you can load all the code for this section into a folder called `intro6` by running `daml new intro6 --template daml-intro-6`

1.11.7.1 Preventing IOU Revocation

The `SimpleIou` contract from *Transform Data Using Choices* and *Add Constraints to a Contract* has one major problem: The contract is only signed by the `issuer`. The signatories are the parties with the power to create and archive contracts. If Alice gave Bob a `SimpleIou` for $100 in exchange for some goods, she could just archive it after receiving the goods. Bob would have a record of such actions, but would have to resort to off-ledger means to get his money back:

```
template SimpleIou
  with
    issuer : Party
    owner : Party
    cash : Cash
  where
    signatory issuer
```

```
simple_iou_test = do
  alice <- allocateParty "Alice"
  bob <- allocateParty "Bob"

  -- Alice and Bob enter into a trade.
  -- Alice transfers the payment as a SimpleIou.
  iou <- submit alice do
    createCmd SimpleIou with
      issuer = alice
      owner = bob
      cash = Cash with
        amount = 100.0
        currency = "USD"

  passTime (days 1)
  -- Bob delivers the goods.

  passTime (minutes 10)
  -- Alice just deletes the payment.
  submit alice do
    archiveCmd iou
```

For a party to have any guarantees that only those transformations specified in the choices are actually followed, they either need to be a signatory themselves, or trust one of the signatories to not agree to transactions that archive and re-create contracts in unexpected ways. To make the `SimpleIou` safe for Bob, you need to add him as a signatory:

```
template Iou
  with
    issuer : Party
    owner : Party
    cash : Cash
  where
    signatory issuer, owner

    choice Transfer
      : ContractId Iou
      with
        newOwner : Party
```

(continues on next page)

(continued from previous page)

```
      controller owner
      do
        assertMsg "newOwner cannot be equal to owner." (owner /= newOwner)
        create this with
          owner = newOwner
```

There's a new problem here: There is no way for Alice to issue or transfer this `Iou` to Bob. To get an `Iou` with Bob's signature as `owner` onto the ledger, his authority is needed:

```
iou_test = do
  alice <- allocateParty "Alice"
  bob <- allocateParty "Bob"

  -- Alice and Bob enter into a trade.
  -- Alice wants to give Bob an Iou, but she can't without Bob's authority.
  submitMustFail alice do
    createCmd Iou with
      issuer = alice
      owner = bob
      cash = Cash with
        amount = 100.0
        currency = "USD"

  -- She can issue herself an Iou.
  iou <- submit alice do
    createCmd Iou with
      issuer = alice
      owner = alice
      cash = Cash with
        amount = 100.0
        currency = "USD"

  -- However, she can't transfer it to Bob.
  submitMustFail alice do
    exerciseCmd iou Transfer with
      newOwner = bob
```

This may seem awkward, but notice that the `ensure` clause is gone from the `Iou` again. The above `Iou` can contain negative values so Bob should be glad that `Alice` cannot put his signature on any `Iou`.

You'll now learn a couple of common ways of building issuance and transfer workflows for the above `Iou`, before diving into the authorization model in full.

1.11.7.2 Use Propose-Accept Workflows for One-Off Authorization

If there is no standing relationship between Alice and Bob, Alice can propose the issuance of an Iou to Bob, giving him the choice to accept. You can do so by introducing a proposal contract `IouProposal`:

```
template IouProposal
  with
    iou : Iou
  where
    signatory iou.issuer
```

(continues on next page)

(continued from previous page)

```
    observer iou.owner

    choice IouProposal_Accept
        : ContractId Iou
        controller iou.owner
        do
          create iou
```

Note how we have used the fact that templates are records here to store the `Iou` in a single field:

```
  iouProposal <- submit alice do
    createCmd IouProposal with
      iou = Iou with
        issuer = alice
        owner = bob
        cash = Cash with
          amount = 100.0
          currency = "USD"

  submit bob do
    exerciseCmd iouProposal IouProposal_Accept
```

The `IouProposal` contract carries the authority of `iou.issuer` by virtue of them being a signatory. By exercising the `IouProposal_Accept` choice, Bob adds his authority to that of Alice, which is why an `Iou` with both signatories can be created in the context of that choice.

The choice is called `IouProposal_Accept`, not `Accept`, because propose-accept patterns are very common. In fact, you'll see another one just below. As each choice defines a record type, you cannot have two choices of the same name in scope. It's a good idea to qualify choice names to ensure uniqueness.

The above solves issuance, but not transfers. You can solve transfers exactly the same way, though, by creating a `TransferProposal`:

```
template IouTransferProposal
  with
    iou : Iou
    newOwner : Party
  where
    signatory (signatory iou)
    observer (observer iou), newOwner

    choice IouTransferProposal_Cancel
      : ContractId Iou
      controller iou.owner
      do
        create iou

    choice IouTransferProposal_Reject
      : ContractId Iou
      controller newOwner
      do
        create iou

    choice IouTransferProposal_Accept
```

(continues on next page)

(continued from previous page)

```
    : ContractId Iou
    controller newOwner
    do
      create iou with
        owner = newOwner
```

In addition to defining the signatories of a contract, `signatory` can also be used to extract the signatories from another contract. Instead of writing `signatory (signatory iou)`, you could write `signatory iou.issuer, iou.owner`.

The `IouProposal` had a single signatory so it could be cancelled easily by archiving it. Without a `Cancel` choice, the `newOwner` could abuse an open TransferProposal as an option. The triple `Accept`, `Reject`, `Cancel` is common to most proposal templates.

To allow an `iou.owner` to create such a proposal, you need to give them the choice to propose a transfer on the `Iou` contract. The choice looks just like the above `Transfer` choice, except that a `IouTransferProposal` is created instead of an `Iou`:

```
  choice ProposeTransfer
    : ContractId IouTransferProposal
    with
      newOwner : Party
    controller owner
    do
      assertMsg "newOwner cannot be equal to owner." (owner /= newOwner)
      create IouTransferProposal with
        iou = this
        newOwner
```

Bob can now transfer his `Iou`. The transfer workflow can even be used for issuance:

```
charlie <- allocateParty "Charlie"

-- Alice issues an Iou using a transfer proposal.
tpab <- submit alice do
  createCmd IouTransferProposal with
    newOwner = bob
    iou = Iou with
      issuer = alice
      owner = alice
      cash = Cash with
        amount = 100.0
        currency = "USD"

-- Bob accepts the transfer from Alice.
iou2 <- submit bob do
  exerciseCmd tpab IouTransferProposal_Accept

-- Bob offers Charlie a transfer.
tpbc <- submit bob do
  exerciseCmd iou2 ProposeTransfer with
    newOwner = charlie

-- Charlie accepts the transfer from Bob.
submit charlie do
  exerciseCmd tpbc IouTransferProposal_Accept
```

1.11.7.3 Use Role Contracts for Ongoing Authorization

Many actions, like the issuance of assets or their transfer, can be pre-agreed. You can represent this succinctly in Daml through relationship or role contracts.

Jointly, an `owner` and `newOwner` can transfer an asset, as demonstrated in the script above. In *Composing Choices*, you will see how to compose the `ProposeTransfer` and `IouTransferProposal_Accept` choices into a single new choice, but for now, here is a different way. You can give them the joint right to transfer an IOU:

```
    choice Mutual_Transfer
      : ContractId Iou
      with
        newOwner : Party
      controller owner, newOwner
      do
        create this with
          owner = newOwner
```

Up to now, the controllers of choices were known from the current contract. Here, the `newOwner` variable is part of the choice arguments, not the `Iou`.

This is also the first time we have shown a choice with more than one controller. If multiple controllers are specified, the authority of *all* the controllers is needed. Here, neither `owner`, nor `newOwner` can execute a transfer unilaterally, hence the name `Mutual_Transfer`.

```
template IouSender
  with
    sender : Party
    receiver : Party
  where
    signatory receiver
    observer sender

    nonconsuming choice Send_Iou
      : ContractId Iou
      with
        iouCid : ContractId Iou
      controller sender
      do
        iou <- fetch iouCid
        assert (iou.cash.amount > 0.0)
        assert (sender == iou.owner)
        exercise iouCid Mutual_Transfer with
          newOwner = receiver
```

The above `IouSender` contract now gives one party, the `sender` the right to send `Iou` contracts with positive amounts to a `receiver`. The `nonconsuming` keyword on the choice `Send_Iou` changes the behaviour of the choice so that the contract it's exercised on does not get archived when the choice is exercised. That way the `sender` can use the contract to send multiple Ious.

Here it is in action:

```
  -- Bob allows Alice to send him Ious.
  sab <- submit bob do
    createCmd IouSender with
```

(continues on next page)

(continued from previous page)

```
    sender = alice
    receiver = bob

-- Charlie allows Bob to send him Ious.
sbc <- submit charlie do
  createCmd IouSender with
    sender = bob
    receiver = charlie

-- Alice can now send the Iou she issued herself earlier.
iou4 <- submit alice do
  exerciseCmd sab Send_Iou with
    iouCid = iou

-- Bob sends it on to Charlie.
submit bob do
  exerciseCmd sbc Send_Iou with
    iouCid = iou4
```

1.11.7.4 Daml's Authorization Model

Hopefully, the above will have given you a good intuition for how authority is passed around in Daml. In this section you'll learn about the formal authorization model to allow you to reason through your contract models. This will allow you to construct them in such a way that you don't run into authorization errors at runtime, or, worse still, allow malicious transactions.

In *Choices In the Ledger Model* you learned that a transaction is, equivalently, a tree of transactions, or a forest of actions, where each transaction is a list of actions, and each action has a child-transaction called its consequences.

Each action has a set of *required authorizers* – the parties that must authorize that action – and each transaction has a set of *authorizers* – the parties that did actually authorize the transaction.

The authorization rule is that the required authorizers of every action are a subset of the authorizers of the parent transaction.

The required authorizers of actions are:

> The required authorizers of an **exercise action** are the controllers on the corresponding choice. Remember that `Archive` and `archive` are just an implicit choice with the signatories as controllers.
> The required authorizers of a **create action** are the signatories of the contract.
> The required authorizers of a **fetch action** (which also includes `fetchByKey`) are somewhat dynamic and covered later.

The authorizers of transactions are:

> The root transaction of a commit is authorized by the submitting party.
> The consequences of an exercise action are authorized by the actors of that action plus the signatories of the contract on which the action was taken.

An Authorization Example

Consider the transaction from the script above where Bob sends an `Iou` to Charlie using a `Send_Iou` contract. It is authorized as follows, ignoring fetches:

> Bob submits the transaction so he's the authorizer on the root transaction.
> The root transaction has a single action, which is to exercise `Send_Iou` on a `IouSender` contract with Bob as `sender` and Charlie as `receiver`. Since the controller of that choice is the `sender`, Bob is the required authorizer.
> The consequences of the `Send_Iou` action are authorized by its actors, Bob, as well as signatories of the contract on which the action was taken. That's Charlie in this case, so the consequences are authorized by both Bob and Charlie.
> The consequences contain a single action, which is a `Mutual_Transfer` with Charlie as `newOwner` on an `Iou` with `issuer` Alice and `owner` Bob. The required authorizers of the action are the `owner`, Bob, and the `newOwner`, Charlie, which matches the parent's authorizers.
> The consequences of `Mutual_Transfer` are authorized by the actors (Bob and Charlie), as well as the signatories on the Iou (Alice and Bob).
> The single action on the consequences, the creation of an Iou with `issuer` Alice and `owner` Charlie has required authorizers Alice and Charlie, which is a proper subset of the parent's authorizers.

You can see the graph of this transaction in the transaction view of the IDE:

```
TX 12 1970-01-01T00:00:00Z (Parties:276:3)
#12:0
│   disclosed to (since): 'Bob' (12), 'Charlie' (12)
└─> 'Bob' exercises Send_Iou on #10:0 (Parties:IouSender)
          with
            iouCid = #11:3
    children:
    #12:1
    │   disclosed to (since): 'Bob' (12), 'Charlie' (12), 'Alice' (12)
    └─> 'Alice' and 'Bob' fetch #11:3 (Parties:Iou)

    #12:2
    │   disclosed to (since): 'Bob' (12), 'Charlie' (12), 'Alice' (12)
    └─> 'Bob' and 'Charlie' exercise Mutual_Transfer on #11:3 (Parties:Iou)
                            with
                              newOwner = 'Charlie'
        children:
        #12:3
        │   disclosed to (since): 'Bob' (12), 'Charlie' (12), 'Alice' (12)
        └─> 'Alice' and 'Charlie' create Parties:Iou
                                  with
                                    issuer = 'Alice';
                                    owner = 'Charlie';
                                    cash =
                                      (Parties:Cash with
                                        currency = "USD"; amount = 100.0000000000)
```

Note that authority is not automatically transferred transitively.

```
template NonTransitive
  with
    partyA : Party
```

(continues on next page)

(continued from previous page)

```
    partyB : Party
  where
    signatory partyA
    observer partyB

    choice TryA
      : ContractId NonTransitive
      controller partyA
      do
        create NonTransitive with
          partyA = partyB
          partyB = partyA

    choice TryB
      : ContractId NonTransitive
        with
          other : ContractId NonTransitive
      controller partyB
      do
        exercise other TryA
```

```
nt1 <- submit alice do
  createCmd NonTransitive with
    partyA = alice
    partyB = bob
nt2 <- submit alice do
  createCmd NonTransitive with
    partyA = alice
    partyB = bob

submitMustFail bob do
  exerciseCmd nt1 TryB with
    other = nt2
```

The consequences of `TryB` are authorized by both Alice and Bob, but the action `TryA` only has Alice as an actor and Alice is the only signatory on the contract.

Therefore, the consequences of `TryA` are only authorized by Alice. Bob's authority is now missing to create the flipped `NonTransitive` so the transaction fails.

1.11.7.5 Next Up

In *Composing Choices* you will put everything you have learned together to build a simple asset holding and trading model akin to that in the *Daml IOU Quickstart Tutorial*. In that context you'll learn a bit more about the `Update` action and how to use it to compose transactions, as well as about privacy on Daml ledgers.

1.11.8 Composing Choices

It's time to put everything you've learned so far together into a complete and secure Daml model for asset issuance, management, transfer, and trading. This application will have capabilities similar to the one in *Daml IOU Quickstart Tutorial*. In the process you will learn about a few more concepts:

Daml projects, packages and modules
Composition of transactions
Observers and stakeholders
Daml's execution model
Privacy

The model in this section is not a single Daml file, but a Daml project consisting of several files that depend on each other.

Hint: Remember that you can load all the code for this section into a folder called `intro7` by running `daml new intro7 --template daml-intro-7`

1.11.8.1 Daml Projects

Daml is organized in projects, packages and modules. A Daml project is specified using a single `daml.yaml` file, and compiles into a package in Daml's intermediate language, or bytecode equivalent, Daml-LF. Each Daml file within a project becomes a Daml module, which is a bit like a namespace. Each Daml project has a source root specified in the `source` parameter in the project's `daml.yaml` file. The package will include all modules specified in `*.daml` files beneath that source directory.

You can start a new project with a skeleton structure using `daml new project-name` in the terminal. A minimal project would contain just a `daml.yaml` file and an empty directory of source files.

Take a look at the `daml.yaml` for the this chapter's project:

```
sdk-version: __VERSION__
name: __PROJECT_NAME__
source: daml
version: 1.0.0
dependencies:
  - daml-prim
  - daml-stdlib
  - daml-script
```

You can generally set `name` and `version` freely to describe your project. `dependencies` does what the name suggests: It includes dependencies. You should always include `daml-prim` and `daml-stdlib`. The former contains internals of compiler and Daml Runtime, the latter gives access to the Daml Standard Library. `daml-script` contains the types and standard library for Daml Script.

You compile a Daml project by running `daml build` from the project root directory. This creates a `dar` file in `.daml/dist/dist/${project_name}-${project_version}.dar`. A `dar` file is Daml's equivalent of a `JAR` file in Java: it's the artifact that gets deployed to a ledger to load the package and its dependencies. `dar` files are fully self-contained in that they contain all dependencies of the main package. More on all of this in *Work with Dependencies*.

1.11.8.2 Project Structure

This project contains an asset holding model for transferable, fungible assets and a separate trade workflow. The templates are structured in three modules: `Intro.Asset`, `Intro.Asset.Role`, and `Intro.Asset.Trade`.

In addition, there are tests in modules `Test.Intro.Asset`, `Test.Intro.Asset.Role`, and `Test.Intro.Asset.Trade`.

All but the last `.`-separated segment in module names correspond to paths relative to the project source directory, and the last one to a file name. The folder structure therefore looks like this:

```
.
├── daml
│   ├── Intro
│   │   ├── Asset
│   │   │   ├── Role.daml
│   │   │   └── Trade.daml
│   │   └── Asset.daml
│   └── Test
│       └── Intro
│           ├── Asset
│           │   ├── Role.daml
│           │   └── Trade.daml
│           └── Asset.daml
└── daml.yaml
```

Each file contains a module header. For example, `daml/Intro/Asset/Role.daml`:

```
module Intro.Asset.Role where
```

You can import one module into another using the `import` keyword. The `LibraryModules` module imports all six modules:

```
import Intro.Asset
```

Imports always have to appear just below the module declaration. You can optionally add a list of names after the import to import only the selected names:

```
import DA.List (sortOn, groupOn)
```

If your module contains any Daml Scripts, you need to import the corresponding functionality:

```
import Daml.Script
```

1.11.8.3 Project Overview

The project both changes and adds to the `Iou` model presented in *Parties and Authority*:

> Assets are fungible in the sense that they have `Merge` and `Split` choices that allow the `owner` to manage their holdings.
> Transfer proposals now need the authorities of both `issuer` and `newOwner` to accept. This makes `Asset` safer than `Iou` from the issuer's point of view.

With the `Iou` model, an `issuer` could end up owing cash to anyone as transfers were authorized by just `owner` and `newOwner`. In this project, only parties having an `AssetHolder` contract can end up owning assets. This allows the `issuer` to determine which parties may own their assets.
The `Trade` template adds a swap of two assets to the model.

1.11.8.4 Composed Choices and Scripts

This project showcases how you can put the `Update` and `Script` actions you learned about in *Parties and Authority* to good use. For example, the `Merge` and `Split` choices each perform several actions in their consequences.

Two create actions in case of `Split`
One create and one archive action in case of `Merge`

```
choice Split
  : SplitResult
  with
    splitQuantity : Decimal
  controller owner
  do
    splitAsset <- create this with
      quantity = splitQuantity
    remainder <- create this with
      quantity = quantity - splitQuantity
    return SplitResult with
      splitAsset
      remainder

choice Merge
  : ContractId Asset
  with
    otherCid : ContractId Asset
  controller owner
  do
    other <- fetch otherCid
    assertMsg
      "Merge failed: issuer does not match"
      (issuer == other.issuer)
    assertMsg
      "Merge failed: owner does not match"
      (owner == other.owner)
    assertMsg
      "Merge failed: symbol does not match"
      (symbol == other.symbol)
    archive otherCid
    create this with
      quantity = quantity + other.quantity
```

The `return` function used in `Split` is available in any `Action` context. The result of `return x` is a no-op containing the value `x`. It has an alias `pure`, indicating that it's a pure value, as opposed to a value with side-effects. The `return` name makes sense when it's used as the last statement in a `do` block as its argument is indeed the return -value of the `do` block in that case.

Taking transaction composition a step further, the `Trade_Settle` choice on `Trade` composes two `exercise` actions:

```
    choice Trade_Settle
      : (ContractId Asset, ContractId Asset)
      with
        quoteAssetCid : ContractId Asset
        baseApprovalCid : ContractId TransferApproval
      controller quoteAsset.owner
      do
        fetchedBaseAsset <- fetch baseAssetCid
        assertMsg
          "Base asset mismatch"
          (baseAsset == fetchedBaseAsset with
            observers = baseAsset.observers)

        fetchedQuoteAsset <- fetch quoteAssetCid
        assertMsg
          "Quote asset mismatch"
          (quoteAsset == fetchedQuoteAsset with
            observers = quoteAsset.observers)

        transferredBaseCid <- exercise
          baseApprovalCid TransferApproval_Transfer with
            assetCid = baseAssetCid

        transferredQuoteCid <- exercise
          quoteApprovalCid TransferApproval_Transfer with
            assetCid = quoteAssetCid

        return (transferredBaseCid, transferredQuoteCid)
```

The resulting transaction, with its two nested levels of consequences, can be seen in the `test_trade` script in `Test.Intro.Asset.Trade`:

```
TX 14 1970-01-01T00:00:00Z (Test.Intro.Asset.Trade:79:23)
#14:0
│   disclosed to (since): 'Alice' (14), 'Bob' (14)
└─> 'Bob' exercises Trade_Settle on #12:0 (Intro.Asset.Trade:Trade)
          with
            quoteAssetCid = #9:1; baseApprovalCid = #13:1
    children:
    #14:1
    │   disclosed to (since): 'Alice' (14), 'Bob' (14), 'USD_Bank' (14)
    └─> 'Alice' and 'USD_Bank' fetch #10:1 (Intro.Asset:Asset)

    #14:2
    │   disclosed to (since): 'Alice' (14), 'Bob' (14), 'EUR_Bank' (14)
    └─> 'Bob' and 'EUR_Bank' fetch #9:1 (Intro.Asset:Asset)

    #14:3
    │   disclosed to (since): 'Alice' (14), 'Bob' (14), 'USD_Bank' (14)
    └─> 'Alice' and 'Bob' exercise TransferApproval_Transfer on #13:1 (Intro.
↪Asset:TransferApproval)
                            with
                              assetCid = #10:1
        children:
        #14:4
        │   disclosed to (since): 'Alice' (14), 'Bob' (14), 'USD_Bank' (14)
```

(continues on next page)

(continued from previous page)

```
        └─> 'Alice' and 'USD_Bank' fetch #10:1 (Intro.Asset:Asset)

        #14:5
        │   disclosed to (since): 'Alice' (14), 'Bob' (14), 'USD_Bank' (14)
        └─> 'Alice' and 'USD_Bank' exercise Archive on #10:1 (Intro.Asset:Asset)

        #14:6
        │   disclosed to (since): 'Alice' (14), 'Bob' (14), 'USD_Bank' (14)
        └─> 'Bob' and 'USD_Bank' create Intro.Asset:Asset
                                 with
                                   issuer = 'USD_Bank';
                                   owner = 'Bob';
                                   symbol = "USD";
                                   quantity = 100.0000000000;
                                   observers = []

    #14:7
    │   disclosed to (since): 'Alice' (14), 'Bob' (14), 'EUR_Bank' (14)
    └─> 'Alice',
        'Bob' exercises TransferApproval_Transfer on #11:1 (Intro.
↪Asset:TransferApproval)
              with
                assetCid = #9:1
        children:
        #14:8
        │   disclosed to (since): 'Alice' (14), 'Bob' (14), 'EUR_Bank' (14)
        └─> 'Bob' and 'EUR_Bank' fetch #9:1 (Intro.Asset:Asset)

        #14:9
        │   disclosed to (since): 'Alice' (14), 'Bob' (14), 'EUR_Bank' (14)
        └─> 'Bob' and 'EUR_Bank' exercise Archive on #9:1 (Intro.Asset:Asset)

        #14:10
        │   disclosed to (since): 'Alice' (14), 'Bob' (14), 'EUR_Bank' (14)
        └─> 'Alice' and 'EUR_Bank' create Intro.Asset:Asset
                                   with
                                     issuer = 'EUR_Bank';
                                     owner = 'Alice';
                                     symbol = "EUR";
                                     quantity = 90.0000000000;
                                     observers = []
```

Similar to choices, you can see how the scripts in this project are built up from each other:

```
test_issuance = do
  setupResult@(alice, bob, bank, aha, ahb) <- setupRoles

  assetCid <- submit bank do
    exerciseCmd aha Issue_Asset
      with
        symbol = "USD"
        quantity = 100.0

  Some asset <- queryContractId bank assetCid
  assert (asset == Asset with
      issuer = bank
```

(continues on next page)

(continued from previous page)

```
      owner = alice
      symbol = "USD"
      quantity = 100.0
      observers = []
        )

  return (setupResult, assetCid)
```

In the above, the `test_issuance` script in `Test.Intro.Asset.Role` uses the output of the `setupRoles` script in the same module.

The same line shows a new kind of pattern matching. Rather than writing `setupResult <- setupRoles` and then accessing the components of `setupResult` using `_1`, `_2`, etc., you can give them names. It's equivalent to writing:

```
setupResult <- setupRoles
case setupResult of
  (alice, bob, bank, aha, ahb) -> ...
```

Just writing `(alice, bob, bank, aha, ahb) <- setupRoles` would also be legal, but `setupResult` is used in the return value of `test_issuance` so it makes sense to give it a name, too. The notation with `@` allows you to give both the whole value as well as its constituents names in one go.

1.11.8.5 Daml's Execution Model

Daml's execution model is fairly easy to understand, but has some important consequences. You can imagine the life of a transaction as follows:

Command Submission A user submits a list of Commands via the Ledger API of a Participant Node, acting as a *Party* hosted on that Node. That party is called the requester.

Interpretation Each Command corresponds to one or more Actions. During this step, the `Update` corresponding to each Action is evaluated in the context of the ledger to calculate all consequences, including transitive ones (consequences of consequences, etc.). The result of this is a complete Transaction. Together with its requestor, this is also known as a Commit.

Blinding On ledgers with strong privacy, projections (see *Privacy*) for all involved parties are created. This is also called *projecting*.

Transaction Submission The Transaction/Commit is submitted to the network.

Validation The Transaction/Commit is validated by the network. Who exactly validates can differ from implementation to implementation. Validation also involves scheduling and collision detection, ensuring that the transaction has a well-defined place in the (partial) ordering of Commits, and no double spends occur.

Commitment The Commit is actually committed according to the commit or consensus protocol of the Ledger.

Confirmation The network sends confirmations of the commitment back to all involved Participant Nodes.

Completion The user gets back a confirmation through the Ledger API of the submitting Participant Node.

The first important consequence of the above is that all transactions are committed atomically. Either a transaction is committed as a whole and for all participants, or it fails.

That's important in the context of the `Trade_Settle` choice shown above. The choice transfers a `baseAsset` one way and a `quoteAsset` the other way. Thanks to transaction atomicity, there is no chance that either party is left out of pocket.

The second consequence is that the requester of a transaction knows all consequences of their submitted transaction – there are no surprises in Daml. However, it also means that the requester must have all the information to interpret the transaction. We also refer to this as Principle 2 a bit later on this page.

That's also important in the context of `Trade`. In order to allow Bob to interpret a transaction that transfers Alice's cash to Bob, Bob needs to know both about Alice's `Asset` contract, as well as about some way for `Alice` to accept a transfer – remember, accepting a transfer needs the authority of `issuer` in this example.

1.11.8.6 Observers

Observers are Daml's mechanism to disclose contracts to other parties. They are declared just like signatories, but using the `observer` keyword, as shown in the `Asset` template:

```
template Asset
  with
    issuer : Party
    owner : Party
    symbol : Text
    quantity : Decimal
    observers : [Party]
  where
    signatory issuer, owner
    ensure quantity > 0.0

    observer observers
```

The `Asset` template also gives the `owner` a choice to set the observers, and you can see how Alice uses it to show her `Asset` to Bob just before proposing the trade. You can try out what happens if she didn't do that by removing that transaction:

```
  usdCid <- submit alice do
    exerciseCmd usdCid SetObservers with
      newObservers = [bob]
```

Observers have guarantees in Daml. In particular, they are guaranteed to see actions that create and archive the contract on which they are an observer.

Since observers are calculated from the arguments of the contract, they always know about each other. That's why, rather than adding Bob as an observer on Alice's `AssetHolder` contract, and using that to authorize the transfer in `Trade_Settle`, Alice creates a one-time authorization in the form of a `TransferAuthorization`. If Alice had lots of counterparties, she would otherwise end up leaking them to each other.

Controllers declared in the `choice` syntax are not automatically made observers, as they can only be calculated at the point in time when the choice arguments are known. On the contrary, controllers declared via the `controller cs can` syntax are automatically made observers, but this syntax is deprecated and will be removed in a future version of Daml.

1.11.8.7 Privacy

Daml's privacy model is based on two principles:

Principle 1. Parties see those actions that they have a stake in. Principle 2. Every party that sees an action sees its (transitive) consequences.

Principle 2 is necessary to ensure that every party can independently verify the validity of every transaction they see.

A party has a stake in an action if

they are a required authorizer of it
they are a signatory of the contract on which the action is performed
they are an observer on the contract, and the action creates or archives it

What does that mean for the `exercise tradeCid Trade_Settle` action from `test_trade`?

Alice is the signatory of `tradeCid` and Bob a required authorizer of the `Trade_Settled` action, so both of them see it. According to principle 2 above, that means they get to see everything in the transaction.

The consequences contain, next to some `fetch` actions, two `exercise` actions of the choice `TransferApproval_Transfer`.

Each of the two involved `TransferApproval` contracts is signed by a different `issuer`, which see the action on their contract. So the EUR_Bank sees the `TransferApproval_Transfer` action for the EUR `Asset` and the USD_Bank sees the `TransferApproval_Transfer` action for the USD `Asset`.

Some Daml ledgers, like the script runner and the Sandbox, work on the principle of data minimization , meaning nothing more than the above information is distributed. That is, the projection of the overall transaction that gets distributed to EUR_Bank in step 4 of *Daml's Execution Model* would consist only of the `TransferApproval_Transfer` and its consequences.

Other implementations, in particular those on public blockchains, may have weaker privacy constraints.

Divulgence

Note that principle 2 of the privacy model means that sometimes parties see contracts that they are not signatories or observers on. If you look at the final ledger state of the `test_trade` script, for example, you may notice that both Alice and Bob now see both assets, as indicated by the Xs in their respective columns:

Intro.Asset:Asset

Alice	Bob	EUR_Bank	USD_Bank	id	status	issuer	owner	symbol	quantity	observers
X	X	-	X	#15:6	active	'USD_Bank'	'Bob'	"USD"	100.0	[]
X	X	X	-	#15:10	active	'EUR_Bank'	'Alice'	"EUR"	90.0	[]

This is because the `create` action of these contracts are in the transitive consequences of the `Trade_Settle` action both of them have a stake in. This kind of disclosure is often called divulgence and needs to be considered when designing Daml models for privacy sensitive applications.

1.11.8.8 Next Up

In *Exception Handling*, we will learn about how errors in your model can be handled in Daml.

1.11.9 Daml Interfaces

After defining a few templates in Daml, you've probably found yourself repeating some behaviors between them. For instance, many templates have a notion of ownership where a party is designated as the owner of the contract, and this party has the power to transfer ownership of the contract to a different party (subject to that party agreeing to the transfer!). Daml Interfaces provide a way to abstract those behaviors into a Daml type.

Hint: Remember that you can load all the code for this section into a folder called `intro13` by running `daml new intro13 --template daml-intro-13`

1.11.9.1 Context

First, define some templates:

```
template Cash
  with
    issuer : Party
    owner : Party
    currency : Text
    amount : Decimal
  where
    signatory issuer, owner
    ensure amount > 0.0

    choice ProposeCashTransfer : ContractId CashTransferProposal
      with newOwner : Party
      controller owner
      do
        create CashTransferProposal with
          cash = this
          newOwner = newOwner
```

```
template CashTransferProposal
  with
    cash : Cash
    newOwner : Party
  where
    signatory (signatory cash)
    observer newOwner

    choice AcceptCashTransferProposal : ContractId Cash
```

(continues on next page)

(continued from previous page)

```
      controller newOwner
      do
        create cash with
          owner = newOwner

    -- Note that RejectCashTransferProposal and WithdrawCashTransferProposal are
    -- almost identical except for the controller - the "recipient" (the new
    -- owner) can reject the proposal, while the "sender" (the old owner) can
    -- withdraw the proposal if the recipient hasn't accepted it already. The
    -- effect in either case is the same: the CashTransferProposal contract is
    -- archived and a new Cash contract is created with the same contents as the
    -- original, but with a new ContractId on the ledger.
    choice RejectCashTransferProposal : ContractId Cash
      controller newOwner
      do
        create cash

    choice WithdrawCashTransferProposal : ContractId Cash
      controller cash.owner
      do
        create cash
```

These declarations from `intro13/daml/Cash.daml` define `Cash` as a simple template with an issuer, an owner, a currency, and an amount. A `Cash` contract grants its `owner` the choice `ProposeCashTransfer`, which allows the owner to propose another party, the `newOwner`, to take over ownership of the asset.

This is mediated by the `CashTransferProposal` template, which grants two choices to the new owner: `AcceptCashTransferProposal` and `RejectCashTransferProposal`, each of which archives the `CashTransferProposal` and creates a new `Cash` contract; in the former case the `owner` of the new `Cash` will be `newOwner`, in the latter, it will be the existing `owner`. Finally, the existing `owner` also has the choice `WithdrawCashTransferProposal`, which archives the proposal and creates a new `Cash` contract with identical contents to the original one.

Overall, the effect is that a `Cash` contract can be transferred to another party, if they agree, in two steps.

The declarations from `intro13/daml/NFT.daml` declare the templates `NFT` and `NFTTransferProposal` following the same pattern, with names changed where appropriate, with the main difference being that an `NFT` has a `url : Text` field whereas `Cash` has `currency : Text` and `amount : Decimal`.

1.11.9.2 Interface Definition

To abstract this behavior, you will next introduce two interfaces: `IAsset` and `IAssetTransferProposal`.

Hint: It is not mandatory to prefix interface names with the letter `I`, but it can be convenient to tell at a glance whether or not a type is an interface.

```
interface IAsset
  where
    viewtype VAsset

    setOwner : Party -> IAsset
    toTransferProposal : Party -> IAssetTransferProposal

    choice ProposeIAssetTransfer : ContractId IAssetTransferProposal
      with newOwner : Party
      controller (view this).owner
      do
        create (toTransferProposal this newOwner)
```

```
interface IAssetTransferProposal
  where
    viewtype VAssetTransferProposal

    asset : IAsset

    choice AcceptIAssetTransferProposal : ContractId IAsset
      controller (view this).newOwner
      do
        create $ setOwner (asset this) (view this).newOwner

    choice RejectIAssetTransferProposal : ContractId IAsset
      controller (view this).newOwner
      do
        create (asset this)

    choice WithdrawIAssetTransferProposal : ContractId IAsset
      controller (view (asset this)).owner
      do
        create (asset this)
```

There are a few things happening here:

1. For each interface, you have defined a `viewtype`. This is mandatory for all interfaces. All viewtypes must be serializable records. The viewtype abstracts the read side by providing a uniform way in which implementations of `IAsset` are represented on the Ledger API. This declaration means that the special `view` method, when applied to a value of this interface, will return the specified type (in this case `VAsset`). This is the definition of `VAsset`:

   ```
   data VAsset = VAsset with
       issuer : Party
       owner : Party
       description : Text
     deriving (Eq, Ord, Show)
   ```

   Hint: See *Serializable Types* for more information on serializability requirements.

2. You have defined the methods `setOwner` and `toTransferProposal` as part of the `IAsset` interface, and method `asset` as part of the `IAssetTransferProposal` interface. Later, when you provide instances of these interfaces, you will see that it is mandatory to implement each of these methods.
3. You have defined the choice `ProposeIAssetTransfer` as part of the `IAsset` interface, and

the choices `AcceptIAssetTransferProposal`, `RejectIAssetTransferProposal` and `WithdrawIAssetTransferProposal` as part of the `IAssetTransferProposal` interface. These correspond one-to-one with the choices of `Cash` / `CashTransferProposal` and `NFT` / `NFTTransferProposal`.
Notice that the choice controller and the choice body are defined in terms of the methods that you bundled with the interfaces, including the special `view` method. For example, the controller of `choice ProposeIAssetTransfer` is `(view this).owner`, that is, it's the `owner` field of the `view` for the implicit current contract `this`, in other words, the owner of the current contract. The body of this choice is `create (toTransferProposal this newOwner)`, so it creates a new contract whose contents are the result of applying the `toTransferProposal` method to the current contract and the `newOwner` field of the choice argument.

Hint: For a detailed explanation of the syntax used here, check out *Reference: Interfaces*

1.11.9.3 Interface Instances

On its own, an interface isn't very useful, since all contracts on the ledger must belong to some template type. In order to make the link between an interface and a template, you must define an interface instance inside the body of either the template or the interface. In this example, add: `interface instance IAsset for Cash` and `interface instance IAssetTransferProposal for CashTransferProposal`:

```
interface instance IAsset for Cash where
  view = VAsset with
    issuer
    owner
    description = show @Cash this

  setOwner newOwner =
    toInterface @IAsset $
      this with
        owner = newOwner

  toTransferProposal newOwner =
    toInterface @IAssetTransferProposal $
      CashTransferProposal with
        cash = this
        newOwner
```

```
interface instance IAssetTransferProposal for CashTransferProposal where
  view = VAssetTransferProposal with
    assetView = view (toInterface @IAsset cash)
    newOwner

  asset = toInterface @IAsset cash
```

The corresponding interface instances for `NFT` and `NFTTransferProposal` are very similar so we omit them here.

Inside the interface instances, you must implement every method defined for the corresponding interface, including the special `view` method. Within each method implementation the variable `this` is in scope, corresponding to the implict current contract, which will have the type of the template

(in this case `Cash` / `CashTransferProposal`), as well as each of the fields of the template type. For example, the `view` definition in `interface instance IAsset for Cash` mentions `issuer` and `owner`, which refer to the issuer and owner of the current `Cash` contract, as well as `this`, which refers to the entire `Cash` contract payload.

The implementations given for each method must match the types given in the interface definition. Notice that the `view` definition discussed above returns a `VAsset`, corresponding to `IAsset`'s `viewtype`. Similarly, `setOwner` returns an `IAsset`, and `toTransferProposal` returns an `IAssetTransferProposal`. In these last two, the function `toInterface` converts values from a template type into an interface type. In `setOwner`, `toInterface` is applied to a `Cash` value (`this with owner = newOwner`), producing an `IAsset` value; in `toTransferProposal`, it is applied to a `CashTransferProposal` value (`CashTransferProposal with {...}`), producing an `IAssetTransferProposal` value.

1.11.9.4 Using an Interface

Now that you have some interfaces and templates with instances for them, you can reduce duplication in the code for different templates by instead going through the common interface.

For instance, both `Cash` and `NFT` are `Assets`, which means that contracts of either template have an owner who can propose to transfer the contract to a third party. Thus, you can use Daml Script (see *Test Templates Using Daml Script*) to test that the same contract can be created by `Alice` and successively transferred to `Bob` and then `Charlie`, who then proposes to transfer to `Dominic`, who rejects the proposal, and finally to `Emily` before withdrawing the proposal, so in the end the contract remains in `Charlie`'s ownership. This procedure is tested on the `Cash` and `NFT` templates by the Daml Script tests `cashTest` and `nftTest`, respectively, both defined in `intro13/daml/Main.daml`.

But that's a lot of duplication! `cashTest` and `nftTest` only differ in the line that creates the original asset and in the names of the choices used. With the new interfaces `IAsset` and `IAssetTransferProposal`, you can write the body of this test a single time, with the name `mkAssetTest`,

```
mkAssetTest assetTxt Parties {..} mkAsset = do
```

You now have not the test itself, but rather a recipe for making the test given some inputs - in this case, `assetTxt` (a label used for debugging), `Parties {..}` (a structure containing the `Party` values for `Alice` and friends) and finally `mkAsset` (a function that returns a contract value of type `t` when given two `Party` arguments - the constraint `Implements t IAsset` means that `t` must be some template with an interface instance for `IAsset`).

Before looking at the body of `mkAssetTest`, notice how you use it to define the new tests `cashAssetTest` and `nftAssetTest`; these are almost identical except for the label and function given in each case to `mkAssetTest`. In effect, you have abstracted those away, so you don't need to include those details in the body of `mkAssetTest`:

```
cashAssetTest : Script (ContractId IAsset)
cashAssetTest = do
  parties <- allocateParties
  mkAssetTest "Cash" parties mkCash
```

```
mkCash : Party -> Party -> Cash
mkCash issuer owner = Cash with
  issuer
```

(continues on next page)

(continued from previous page)

```
  owner
  currency = "USD"
  amount = 42.0
```

```
nftAssetTest : Script (ContractId IAsset)
nftAssetTest = do
  parties <- allocateParties
  mkAssetTest "NFT" parties mkNft
```

```
mkNft : Party -> Party -> NFT
mkNft issuer owner = NFT with
  issuer
  owner
  url = "https://nyan.feline/"
```

In turn, `mkAssetTest` isn't very different from other Daml `Script`s you might have written before: it uses `do` notation as usual, including `submit` blocks constructed from `Command`s that define the ordered transactions that take place in the test. The main difference is that when querying values of interface types you cannot use the functions `query` and `queryContractId`; instead you must use `queryInterface` (for obtaining the set of visible active contracts of a given interface type) and `queryInterfaceContractId` (for obtaining a single contract given its `ContractId`). Importantly, these functions return the *view* of the contract corresponding to the used interface, rather than the contract record itself. This is because the ledger might contain contracts of template types that you don't know about but that do implement our interface, so the view is the only sensible thing that can be returned by the ledger.

Also note that immediately after creating the asset with `createCmd`, you convert the resulting `ContractId t` into a `ContractId IAsset` using `toInterfaceContractId`, which allows you to exercise `IAsset` choices on it.

```
mkAssetTest : forall t.
  (Template t, Implements t IAsset, HasAgreement t) =>
  Text -> Parties -> (Party -> Party -> t) -> Script (ContractId IAsset)
mkAssetTest assetTxt Parties {..} mkAsset = do
  aliceAsset <-
    alice `submit` do
      toInterfaceContractId @IAsset <$>
        createCmd (mkAsset alice alice)

  aliceAssetView <-
    queryInterfaceContractId @IAsset alice aliceAsset

  debugRaw $ unlines
    [ "Alice's Asset (" <> assetTxt <> "):"
    , "\tContractId: " <> show aliceAsset
    , "\tValue: " <> show aliceAssetView
    ]

  bobAssetTransferProposal <-
    alice `submit` do
      exerciseCmd aliceAsset ProposeIAssetTransfer with
        newOwner = bob

  bobAsset <-
```

(continues on next page)

(continued from previous page)

```
  bob `submit` do
    exerciseCmd bobAssetTransferProposal AcceptIAssetTransferProposal

charlieAssetTransferProposal <-
  bob `submit` do
    exerciseCmd bobAsset ProposeIAssetTransfer with
      newOwner = charlie

charlieAsset <-
  charlie `submit` do
    exerciseCmd charlieAssetTransferProposal AcceptIAssetTransferProposal

dominicAssetTransferProposal <-
  charlie `submit` do
    exerciseCmd charlieAsset ProposeIAssetTransfer with
      newOwner = dominic

charlieAsset' <-
  dominic `submit` do
    exerciseCmd dominicAssetTransferProposal RejectIAssetTransferProposal

emilyAssetTransferProposal <-
  charlie `submit` do
    exerciseCmd charlieAsset' ProposeIAssetTransfer with
      newOwner = emily

charlieAsset'' <-
  charlie `submit` do
    exerciseCmd emilyAssetTransferProposal WithdrawIAssetTransferProposal

charlieAssetView <-
  queryInterfaceContractId @IAsset charlie charlieAsset''

debugRaw $ unlines
  [ "Charlie's Asset (" <> assetTxt <> "):"
  , "\tContractId: " <> show charlieAsset''
  , "\tView: " <> show charlieAssetView
  ]

charlieAssetView ===
  Some (view (toInterface @IAsset (mkAsset alice charlie)))

pure charlieAsset''
```

1.11.10 Exception Handling

The default behavior in Daml is to abort the transaction on any error and roll back all changes that have happened until then. However, this is not always appropriate. In some cases, it makes sense to recover from an error and continue the transaction instead of aborting it.

One option for doing that is to represent errors explicitly via `Either` or `Option` as shown in *Data Types*. This approach has the advantage that it is very explicit about which operations are allowed to fail without aborting the entire transaction. However, it also has two major downsides. First, it can be invasive for operations where aborting the transaction is often the desired behavior, e.g., changing

division to return `Either` or an `Option` to handle division by zero would be a very invasive change and many call sites might not want to handle the error case explicitly. Second, and more importantly, this approach does not allow rolling back ledger actions that have happened before the point where failure is detected; if a contract got created before we hit the error, there is no way to undo that except for aborting the entire transaction (which is what we were trying to avoid in the first place).

By contrast, exceptions provide a way to handle certain types of errors in such a way that, on the one hand, most of the code that is allowed to fail can be written just like normal code, and, on the other hand, the programmer can clearly delimit which part of the current transaction should be rolled back on failure. All of that still happens within the same transaction and is thereby atomic contrary to handling the error outside of Daml.

Hint: Remember that you can load all the code for this section into a folder called `intro8` by running `daml new intro8 --template daml-intro-8`

Our example for the use of exceptions will be a simple shop template. Users can order items by calling a choice and transfer money (in the form of an Iou issued by their bank) from their account to the owner in return.

First, we need to setup a template to represent the account of a user:

```
template Account with
    issuer : Party
    owner : Party
    amount : Decimal
  where
    signatory issuer, owner
    ensure amount > 0.0
    key (issuer, owner) : (Party, Party)
    maintainer key._2

    choice Transfer : () with
        newOwner : Party
        transferredAmount : Decimal
      controller owner, newOwner
      do create this with amount = amount - transferredAmount
         create Iou with issuer = issuer, owner = newOwner, amount =
↪transferredAmount
         pure ()
```

Note that the template has an `ensure` clause that ensures that the amount is always positive so `Transfer` cannot transfer more money than is available.

The shop is represented as a template signed by the owner. It has a field to represent the bank accepted by the owner, a list of observers that can order items, and a fixed price for the items that can be ordered:

```
template Shop
  with
    owner : Party
    bank : Party
    observers : [Party]
    price : Decimal
  where
```

(continues on next page)

(continued from previous page)

```
    signatory owner
    observer observers
```

Note: In a real setting the price of each item for sale might be defined in a separate contract.

The ordering process is then represented by a non-consuming choice on this template which calls `Transfer` and creates an `Order` contract in return:

```
    nonconsuming choice OrderItem : ContractId Order
      with
        shopper : Party
      controller shopper
      do exerciseByKey @Account (bank, shopper) (Transfer owner price)
         create Order
           with
             shopOwner = owner
             shopper = shopper
```

However, the shop owner has realized that often orders fail because the account of their users is not topped up. They have a small trusted userbase they know well so they decide that if the account is not topped up, the shoppers can instead issue an Iou to the owner and pay later. While it would be possible to check the conditions under which `Transfer` will fail in `OrderItem` this can be quite fragile: In this example, the condition is relatively simple but in larger projects replicating the conditions outside the choice and keeping the two in sync can be challenging.

Exceptions allow us to handle this differently. Rather than replicating the checks in `Transfer`, we can instead catch the exception thrown on failure. To do so we need to use a try-catch block. The `try` block defines the scope within which we want to catch exceptions while the `catch` clauses define which exceptions we want to catch and how we want to handle them. In this case, we want to catch the exception thrown by a failed `ensure` clause. This exception is defined in `daml-stdlib` as `PreconditionFailed`. Putting it together our order process for trusted users looks as follows:

```
    nonconsuming choice OrderItemTrusted : ContractId Order
      with
        shopper : Party
      controller shopper
      do cid <- create Order
           with
             shopOwner = owner
             shopper = shopper
         try do
           exerciseByKey @Account (bank, shopper) (Transfer owner price)
         catch
           PreconditionFailed _ -> do
             create Iou with
               issuer = shopper
               owner = owner
               amount = price
             pure ()
         pure cid
```

Let's walk through this code. First, as mentioned, the shop owner is the trusting kind, so he wants to start by creating the `Order` no matter what. Next, he tries to charge the customer for the order.

We could, at this point, check their balance against the cost of the order, but that would amount to duplicating the logic already present in `Account`. This logic is pretty simple in this case, but duplicating invariants is a bad habit to get into. So, instead, we just *try* to charge the account. If that succeeds, we just merrily ignore the entire `catch` clause; if that fails, however, we do not want to destroy the Order contract we had already created. Instead, we want to *catch* the error thrown by the `ensure` clause of `Account` (in this case, it is of type `PreconditionFailed`) and try something else: create an `Iou` contract to register the debt and move on.

Note that if the `Iou` creation still failed (unlikely with our definition of `Iou` here, but could happen in more complex scenarios), because that one is not wrapped in a `try` block, we would revert to the default Daml behaviour and the `Order` creation *would* be rolled back.

In addition to catching built-in exceptions like `PreconditionFailed`, you can also define your own exception types which can be caught and thrown. As an example, let's consider a variant of the `Transfer` choice that only allows for transfers up to a given limit. If the amount is higher than the limit, we throw an exception called `TransferLimitExceeded`.

We first have to define the exception and define a way to represent it as a string. In this case, our exception should store the amount that someone tried to transfer as well as the limit.

```
exception TransferLimitExceeded
  with
    limit : Decimal
    attempted : Decimal
  where
    message "Transfer of " <> show attempted <> " exceeds limit of " <> show limit
```

To throw our own exception, you can use `throw` in `Update` and `Script` or `throwPure` in other contexts.

```
    choice TransferLimited : () with
        newOwner : Party
        transferredAmount : Decimal
      controller owner, newOwner
      do let limit = 50.0
         when (transferredAmount > limit) $
           throw TransferLimitExceeded with
             limit = limit
             attempted = transferredAmount
         create this with amount = amount - transferredAmount
         create Iou with issuer = issuer, owner = newOwner, amount =
↪transferredAmount
         pure ()
```

Finally, we can adapt our choice to catch this exception as well:

```
    nonconsuming choice OrderItemTrustedLimited : ContractId Order
      with
        shopper : Party
      controller shopper
      do try do
           exerciseByKey @Account (bank, shopper) (TransferLimited owner price)
           pure ()
         catch
           PreconditionFailed _ -> do
             create Iou with
```

(continues on next page)

(continued from previous page)

```
            issuer = shopper
            owner = owner
            amount = price
          pure ()
        TransferLimitExceeded _ _ -> do
          create Iou with
            issuer = shopper
            owner = owner
            amount = price
          pure ()
      create Order
        with
          shopOwner = owner
          shopper = shopper
```

For more information on exceptions, take a look at the *language reference*.

1.11.10.1 Next Up

We have now seen how to develop safe models and how we can handle errors in those models in a robust and simple way. But the journey doesn't stop there. In *Work with Dependencies* you will learn how to extend an already running application to enhance it with new features. In that context you'll learn a bit more about the architecture of Daml, about dependencies, and about identifiers.

1.11.11 Work with Dependencies

The application from *Composing Choices* is a complete and secure model for atomic swaps of assets, but there is plenty of room for improvement. However, one can't implement all features before going live with an application so it's important to understand how to change already running code. There are fundamentally two types of change one may want to make:

1. Upgrades, which change existing logic. For example, one might want the `Asset` template to have multiple signatories.
2. Extensions, which merely add new functionality through additional templates.

Upgrades are covered in their own section outside this introduction to Daml: *Upgrading and Extending Daml Applications* so in this section we will extend the *Composing Choices* model with a simple second workflow: a multi-leg trade. In doing so, you'll learn about:

The software architecture of the Daml Stack
Dependencies and Data Dependencies
Identifiers

Since we are extending *Composing Choices*, the setup for this chapter is slightly more complex:

1. In a base directory, load the *Composing Choices* project using `daml new intro7 --template daml-intro-7`. The directory `intro7` here is important as it'll be referenced by the other project we are creating.
2. In the same directory, load this chapter's project using `daml new intro9 --template daml-intro-9`.

`Dependencies` contains a new module `Intro.Asset.MultiTrade` and a corresponding test module `Test.Intro.Asset.MultiTrade`.

1.11.11.1 DAR, DALF, Daml-LF, and the Engine

In *Composing Choices* you already learnt a little about projects, Daml-LF, DAR files, and dependencies. In this chapter we will actually need to have dependencies from the current project to the *Composing Choices* project so it's time to learn a little more about all this.

Let's have a look inside the DAR file of *Composing Choices*. DAR files, like Java JAR files, are just ZIP archives, but the SDK also has a utility to inspect DARs out of the box:

1. Navigate into the `intro7` directory.
2. Build using `daml build -o assets.dar`
3. Run `daml damlc inspect-dar assets.dar`

You'll get a whole lot of output. Under the header DAR archive contains the following files: you'll see that the DAR contains:

1. `*.dalf` files for the project and all its dependencies
2. The original Daml source code
3. `*.hi` and `*.hie` files for each `*.daml` file
4. Some meta-inf and config files

The first file is something like `intro7-1.0.0-887056cbb313b94ab9a6caf34f7fe4fbfe19cb0c861e50d1` `dalf` which is the actual compiled package for the project. `*.dalf` files contain Daml-LF, which is Daml's intermediate language. The file contents are a binary encoded protobuf message from the daml-lf schema. Daml-LF is evaluated on the Ledger by the Daml Engine, which is a JVM component that is part of tools like the IDE's Script runner, the Sandbox, or proper production ledgers. If Daml-LF is to Daml what Java Bytecode is to Java, the Daml Engine is to Daml what the JVM is to Java.

1.11.11.2 Hashes and Identifiers

Under the heading DAR archive contains the following packages: you get a similar looking list of package names, paired with only the long random string repeated. That hexadecimal string, `887056cbb313b94ab9a6caf34f7fe4fbfe19cb0c861e50d1594c665567ab7625` in this case, is the package hash and the primary and only identifier for a package that's guaranteed to be available and preserved. Meta information like name (intro7) and version (1.0.0) help make it human readable but should not be relied upon. You may not always get DAR files from your compiler, but be loading them from a running Ledger, or get them from an artifact repository.

We can see this in action. When a DAR file gets deployed to a ledger, not all meta information is preserved.

1. Note down your main package hash from running `inspect-dar` above
2. Start the project using `daml start`
3. Open a second terminal and run `daml ledger fetch-dar --host localhost --port 6865 --main-package-id "887056cbb313b94ab9a6caf34f7fe4fbfe19cb0c861e50d1594c665 -o assets_ledger.dar`, making sure to replace the hash with the appropriate one.
4. Run `daml damlc inspect-dar assets_ledger.dar`

You'll notice two things. Firstly, a lot of the dependencies have lost their names, they are now only identifiable by hash. We could of course also create a second project `intro7-1.0.0` with completely different contents so even when name and version are available, package hash is the only safe identifier.

That's why over the Ledger API, all types, like templates and records are identified by the triple `(entity name, module name, package hash)`. Your client application should know the package

hashes it wants to interact with. To aid that, `inspect-dar` also provides a machine-readable format for the information it emits: `daml damlc inspect-dar --json assets_ledger.dar`. The `main_package_id` field in the resulting JSON payload is the package hash of our project.

Secondly, you'll notice that all the `*.daml`, `*.hi` and `*.hie` files are gone. This leads us to data dependencies.

1.11.11.3 Dependencies and Data Dependencies

Dependencies under the `daml.yaml` `dependencies` group rely on the `*.hi` files. The information in these files is crucial for dependencies like the Standard Library, which provide functions, types and typeclasses.

However, as you can see above, this information isn't preserved. Furthermore, preserving this information may not even be desirable. Imagine we had built `intro7` with SDK 1.100.0, and are building `intro9` with SDK 1.101.0. All the typeclasses and instances on the inbuilt types may have changed and are now present twice – once from the current SDK and once from the dependency. This gets messy fast, which is why the SDK does not support `dependencies` across SDK versions. For dependencies on contract models that were fetched from a ledger, or come from an older SDK version, there is a simpler kind of dependency called `data-dependencies`. The syntax for `data-dependencies` is the same, but they only rely on the binary `*.dalf` files. The name tries to confer that the main purpose of such dependencies is to handle data: Records, Choices, Templates. The stuff one needs to use contract composability across projects.

For an extension model like this one,``data-dependencies`` are appropriate, so the current project includes *Composing Choices* that way:

```
  - daml-script
data-dependencies:
  - ../intro7/assets.dar
```

You'll notice a module `Test.Intro.Asset.TradeSetup`, which is almost a carbon copy of the *Composing Choices* trade setup Scripts. `data-dependencies` is designed to use existing contracts and data types. Daml Script is not imported. In practice, we also shouldn't expect that the DAR file we download from the ledger using `daml ledger fetch-dar` contains test scripts. For larger projects it's good practice to keep them separate and only deploy templates to the ledger.

1.11.11.4 Structuring Projects

As you've seen here, identifiers depend on the package as a whole and packages always bring all their dependencies with them. Thus changing anything in a complex dependency graph can have significant repercussions. It is therefore advisable to keep dependency graphs simple, and to separate concerns which are likely to change at different rates into separate packages.

For example, in all our projects in this intro, including this chapter, our scripts are in the same project as our templates. In practice, that means changing a test changes all identifiers, which is not desirable. It's better for maintainability to separate tests from main templates. If we had done that in *Composing Choices*, that would also have saved us from copying *Composing Choices*.

Similarly, we included `Trade` in the same project as `Asset` in *Composing Choices*, even though `Trade` is a pure extension to the core `Asset` model. If we expect `Trade` to need more frequent changes, it may be a good idea to split it out into a separate project from the start.

1.11.11.5 Next Up

The `MultiTrade` model has more complex control flow and data handling than previous models. In *Functional Programming 101* you'll learn how to write more advanced logic: control flow, folds, common typeclasses, custom functions, and the Standard Library. We'll be using the same projects so don't delete your folders just yet.

1.11.12 Functional Programming 101

In this chapter, you will learn more about expressing complex logic in a functional language like Daml. Specifically, you'll learn about

Function signatures and functions
Advanced control flow (`if...else`, folds, recursion, `when`)

If you no longer have your *Composing Choices* and *Work with Dependencies* projects set up, and want to look back at the code, please follow the setup instructions in *Work with Dependencies* to get hold of the code for this chapter.

Note: There is a project template `daml-intro-10` for this chapter, but it only contains a single source file with the code snippets embedded in this section.

1.11.12.1 The Haskell Connection

The previous chapters of this introduction to Daml have mostly covered the structure of templates, and their connection to the *Daml Ledger Model*. The logic of what happens within the `do` blocks of choices has been kept relatively simple. In this chapter, we will dive deeper into Daml's expression language, the part that allows you to write logic inside those `do` blocks. But we can only scratch the surface here. Daml borrows a lot of its language from Haskell. If you want to dive deeper, or learn about specific aspects of the language you can refer to standard literature on Haskell. Some recommendations:

Finding Success and Failure in Haskell (Julie Maronuki, Chris Martin)
Haskell Programming from first principles (Christopher Allen, Julie Moronuki)
Learn You a Haskell for Great Good! (Miran Lipova a)
Programming in Haskell (Graham Hutton)
Real World Haskell (Bryan O'Sullivan, Don Stewart, John Goerzen)

When comparing Daml to Haskell it's worth noting:

Haskell is a lazy language, which allows you to write things like `head [1..]`, meaning take the first element of an infinite list . Daml by contrast is strict. Expressions are fully evaluated, which means it is not possible to work with infinite data structures.
Daml has a `with` syntax for records and dot syntax for record field access, neither of which is present in Haskell. However, Daml supports Haskell's curly brace record notation.
Daml has a number of Haskell compiler extensions active by default.
Daml doesn't support all features of Haskell's type system. For example, there are no existential types or GADTs.
Actions are called Monads in Haskell.

1.11.12.2 Functions

In *Data Types* you learnt about one half of Daml's type system: Data types. It's now time to learn about the other, which are Function types. Function types in Daml can be spotted by looking for `->` which can be read as maps to .

For example, the function signature `Int -> Int` maps an integer to another integer. There are many such functions, but one would be:

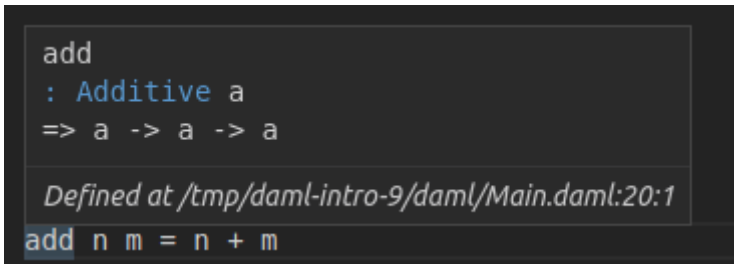
```
increment : Int -> Int
increment n = n + 1
```

You can see here that the function declaration and the function definitions are separate. The declaration can be omitted in cases where the type can be inferred by the compiler, but for top-level functions (ie ones at the same level as templates, directly under a module), it's often a good idea to include them for readability.

In the case of `increment` it could have been omitted. Similarly, we could define a function `add` without a declaration:

```
add n m = n + m
```

If you do this, and wonder what type the compiler has inferred, you can hover over the function name in the IDE:



What you see here is a slightly more complex signature:

```
add : Additive a => a -> a -> a
```

There are two interesting things going on here:

1. We have more than one `->`.
2. We have a type parameter `a` with a constraint `Additive a`.

Function Application

Let's start by looking at the right hand part `a -> a -> a`. The `->` is right associative, meaning `a -> a -> a` is equivalent to `a -> (a -> a)`. Using the maps to way of reading `->`, we get `a` maps to a function that maps `a` to `a` .

And this is indeed what happens. We can define a different version of `increment` by *partially applying* `add`:

```
increment2 = add 1
```

If you try this out in your IDE, you'll see that the compiler infers type `Int -> Int` again. It can do so because of the literal `1 : Int`.

So if we have a function `f : a -> b -> c -> d` and a value `valA : a`, we get `f valA : b -> c -> d`, i.e. we can apply the function argument by argument. If we also had `valB : b`, we would have `f valA valB : c -> d`. What this tells you is that function *application* is left associative: `f valA valB == (f valA) valB`.

Infix Functions

Now `add` is clearly just an alias for `+`, but what is `+`? `+` is just a function. It's only special because it starts with a symbol. Functions that start with a symbol are *infix* by default which means they can be written between two arguments. That's why we can write `1 + 2` rather than `+ 1 2`. The rules for converting between normal and infix functions are simple. Wrap an infix function in parentheses to use it as a normal function, and wrap a normal function in backticks to make it infix:

```
three = 1 `add` 2
```

With that knowledge, we could have defined `add` more succinctly as the alias that it is:

```
add2 : Additive a => a -> a -> a
add2 = (+)
```

If we want to partially apply an infix operation we can also do that as follows:

```
increment3 = (1 +)
double = (* 2)
```

Note: While function application is left associative by default, infix operators can be declared left or right associative and given a precedence. Good examples are the boolean operations `&&` and `||`, which are declared right associative with precedences 3 and 2, respectively. This allows you to write `True || True && False` and get value `True`. See section 4.4.2 of the Haskell 98 report for more on fixities.

Type Constraints

The `Additive a =>` part of the signature of `add` is a type constraint on the type parameter `a`. `Additive` here is a typeclass. You already met typeclasses like `Eq` and `Show` in *Data Types*. The `Additive` typeclass says that you can add a thing, i.e. there is a function `(+) : a -> a -> a`. Now the way to read the full signature of `add` is Given that `a` has an instance for the `Additive` typeclass, `a` maps to a function which maps `a` to `a` .

Typeclasses in Daml are a bit like interfaces in other languages. To be able to add two things using the `+` function, those things need to expose (have an instance for) the `Additive` interface (typeclass).

Unlike interfaces, typeclasses can have multiple type parameters. A good example, which also demonstrates the use of multiple constraints at the same time, is the signature of the `exercise` function:

```
exercise : (Template t, Choice t c r) => ContractId t -> c -> Update r
```

Let's turn this into prose: Given that `t` is the type of a template, and that `t` has a choice `c` with return type `r`, the `exercise` function maps a `ContractId` for a contract of type `t` to a function that takes the choice arguments of type `c` and returns an `Update` resulting in type `r`.

That's quite a mouthful, and does require one to know what *meaning* the typeclass `Choice` gives to parameters `t c` and `r`, but in many cases, that's obvious from the context or names of typeclasses and variables.

Using single letters, while common, is not mandatory. The above may be made a little bit clearer by expanding the type parameter names, at the cost of making the code a bit longer:

```
exercise : (Template template, Choice template choice result) =>
             ContractId template -> choice -> Update result
```

Pattern Matching in Arguments

You met pattern matching in *Data Types*, using `case` expressions which is one way of pattern matching. However, it can also be convenient to do the pattern matching at the level of function arguments. Think about implementing the function `uncurry`:

```
uncurry : (a -> b -> c) -> (a, b) -> c
```

`uncurry` takes a function with two arguments (or more, since `c` could be a function), and turns it into a function from a 2-tuple to `c`. Here are three ways of implementing it, using tuple accessors, `case` pattern matching, and function pattern matching:

```
uncurry1 f t = f t._1 t._2

uncurry2 f t = case t of
  (x, y) -> f x y

uncurry f (x, y) = f x y
```

Any pattern matching you can do in `case` you can also do at the function level, and the compiler helpfully warns you if you did not cover all cases, which is called non-exhaustive .

```
fromSome : Optional a -> a
fromSome (Some x) = x
```

The above will give you a warning:

```
warning:
  Pattern match(es) are non-exhaustive
  In an equation for 'fromSome': Patterns not matched: None
```

A function that does not cover all its cases, like `fromSome` here, is called a *partial* function. `fromSome None` will cause a runtime error.

We can use function level pattern matching together with a feature called *Record Wildcards* to write the function `issueAsset` in *Work with Dependencies*:

```
issueAsset : Asset -> Script (ContractId Asset)
issueAsset asset@(Asset with ..) = do
  assetHolders <- queryFilter @AssetHolder issuer
    (\ah -> (ah.issuer == issuer) && (ah.owner == owner))

  case assetHolders of
    (ahCid, _)::_ -> submit asset.issuer do
      exerciseCmd ahCid Issue_Asset with ..
    [] -> abort ("No AssetHolder found for " <> show asset)
```

The `..` in the pattern match here means bind all fields from the given record to local variables, so we have local variables `issuer`, `owner`, etc.

The `..` in the second to last line means fill all fields of the new record using local variables of the matching names, in this case (per the definition of `Issue_Asset`), `symbol` and `quantity`, taken from the `asset` argument to the function. In other words, this is equivalent to:

```
exerciseCmd ahCid Issue_Asset with symbol = asset.symbol, quantity = asset.
↪quantity
```

because the notation `asset@(Asset with ..)` binds `asset` to the entire record, while also binding all of the fields of `asset` to local variables.

Functions Everywhere

You have probably already guessed it: Anywhere you can put a value in Daml you can also put a function. Even inside data types:

```
data Predicate a = Predicate with
  test : a -> Bool
```

More often it makes sense to define functions locally, inside a `let` clause or similar. Good examples of this are the `validate` and `transfer` functions defined locally in the `Trade_Settle` choice of the model from *Work with Dependencies*:

```
        let
          validate (asset, assetCid) = do
            fetchedAsset <- fetch assetCid
            assertMsg
              "Asset mismatch"
              (asset == fetchedAsset with
                observers = asset.observers)

        mapA_ validate (zip baseAssets baseAssetCids)
        mapA_ validate (zip quoteAssets quoteAssetCids)

        let
          transfer (assetCid, approvalCid) = do
            exercise approvalCid TransferApproval_Transfer with assetCid

        transferredBaseCids <- mapA transfer (zip baseAssetCids baseApprovalCids)
        transferredQuoteCids <- mapA transfer (zip quoteAssetCids
↪quoteApprovalCids)
```

You can see that the function signature is inferred from the context here. If you look closely (or hover over the function in the IDE), you'll see that it has signature

```
validate : (HasFetch r, Eq r, HasField "observers" r a) => (r, ContractId r) ->
↪Update ()
```

Note: Bear in mind that functions are not serializable, so you can't use them inside template arguments, as choice inputs, or as choice outputs. They also don't have instances of the `Eq` or `Show` typeclasses which one would commonly want on data types.

The `mapA` and `mapA_` functions loop through the lists of assets and approvals and apply the functions `validate` and `transfer` to each element of those lists, performing the resulting `Update` action in the process. We'll look at that more closely under *Looping* below.

Lambdas

Daml supports inline functions, called lambda s. They are defined using the `(\x y z -> ...)` syntax. For example, a lambda version of `increment` would be `(\n -> n + 1)`.

1.11.12.3 Control Flow

In this section, we will cover branching and looping, and look at a few common patterns of how to translate procedural code into functional code.

Branching

Until *Composing Choices* the only real kind of control flow introduced has been `case`, which is a powerful tool for branching.

If ... Else

Add Constraints to a Contract also showed a seemingly self-explanatory `if ... else` expression, but didn't explain it further. Let's implement the function `boolToInt : Bool -> Int` which in typical fashion maps `True` to `1` and `False` to `0`. Here is an implementation using `case`:

```
boolToInt b = case b of
  True -> 1
  False -> 0
```

If you write this function in the IDE, you'll get a warning from the linter:

```
Suggestion: Use if
Found:
case b of
    True -> 1
    False -> 0
Perhaps:
if b then 1 else 0
```

The linter knows the equivalence and suggests a better implementation:

```
boolToInt2 b = if b
  then 1
  else 0
```

In short: `if ... else` expressions are equivalent to `case` expressions, but can be easier to read.

Control Flow as Expressions

`case` and `if ... else` expressions really are control flow in the sense that they short-circuit:

```
doError t = case t of
  "True" -> True
  "False" -> False
  _ -> error ("Not a Bool: " <> t)
```

This function behaves as you would expect: the error only gets evaluated if an invalid text is passed in.

This is different from functions, where all arguments are evaluated immediately:

```
ifelse b t e = if b then t else e
boom = ifelse True 1 (error "Boom")
```

In the above, `boom` is an error.

While providing proper control flow, `case` and `if ... else` expressions do result in a value when evaluated. You can actually see that in the function definitions above. Since each of the functions is defined just as a `case` or `if ... else` expression, the value of the evaluated function is just the value of the `case` or `if ... else` expression. Values have a type: the `if ... else` expression in `boolToInt2` has type `Int` as that is what the function returns; similarly, the `case` expression in `doError` has type `Bool`. To be able to give such expressions an unambiguous type, each branch needs to have the same type. The below function does not compile as one branch tries to return an `Int` and the other a `Text`:

```
typeError b = if b
  then 1
  else "a"
```

If we need functions that can return two (or more) types of things we need to encode that in the return type. For two possibilities, it's common to use the `Either` type:

```
intOrText : Bool -> Either Int Text
intOrText b = if b
  then Left 1
  else Right "a"
```

When you have more than two possible types (and sometimes even just for two types), it can be clearer to define your own variant type to wrap all possibilities.

Branching in Actions

The most common case where this becomes important is inside `do` blocks. Say we want to create a contract of one type in one case, and of another type in another case. Let's say we have two template types and want to write a function that creates an `S` if a condition is met, and a `T` otherwise.

```
template T
  with
    p : Party
  where
    signatory p

template S
  with
    p : Party
  where
    signatory p
```

It would be tempting to write a simple `if ... else`, but it won't typecheck if each branch returns a different type:

```
typeError b p = if b
  then create T with p
  else create S with p
```

We have two options:

1. Use the `Either` trick from above.
2. Get rid of the return types.

```
ifThenSElseT1 b p = if b
  then do
    cid <- create S with p
    return (Left cid)
  else do
    cid <- create T with p
    return (Right cid)

ifThenSElseT2 b p = if b
  then do
    create S with p
    return ()
  else do
    create T with p
    return ()
```

The latter is so common that there is a utility function in `DA.Action` to get rid of the return type: `void : Functor f => f a -> f ()`.

```
ifThenSElseT3 b p = if b
  then void (create S with p)
  else void (create T with p)
```

`void` also helps express control flow of the type Create a `T` only if a condition is met.

```
conditionalS b p = if b
  then void (create S with p)
  else return ()
```

Note that we still need the `else` clause of the same type `()`. This pattern is so common, it's encapsulated in the standard library function `DA.Action.when : (Applicative f) => Bool -> f () -> f ()`.

```
conditionalS2 b p = when b (void (create S with p))
```

Despite `when` looking like a simple function, the compiler does some magic so that it short-circuits evaluation just like `if ... else` and `case`. The following `noop` function is a no-op (i.e. does nothing), not an error as one might otherwise expect:

```
noop : Update () = when False (error "Foo")
```

With `case`, `if ... else`, `void` and `when`, you can express all branching. However, one additional feature you may want to learn is guards. They are not covered here, but can help avoid deeply nested `if ... else` blocks. Here's just one example. The Haskell sources at the beginning of the chapter cover this topic in more depth.

```
tellSize : Int -> Text
tellSize d
  | d < 0 = "Negative"
  | d == 0 = "Zero"
  | d == 1 = "Non-Zero"
  | d < 10 = "Small"
  | d < 100 = "Big"
  | d < 1000 = "Huge"
  | otherwise = "Enormous"
```

Looping

Other than branching, the most common form of control flow is looping. Looping is usually used to iteratively modify some state. We'll use JavaScript in this section to illustrate the procedural way of doing things.

```
function sum(intArr) {
  var result = 0;
  intArr.forEach (i => {
    result += i;
  });
  return result;
}
```

A more general loop looks like this:

```
function whileF(init, cont, step, finalize) {
  var state = init();
  while (cont(state)) {
    state = step(state);
  }
```

(continues on next page)

(continued from previous page)

```
  return finalize(state);
}
```

In both cases, state is being mutated: `result` in the former, `state` in the latter. Values in Daml are immutable, so it needs to work differently. In Daml we will do this with folds and recursion.

Folds

Folds correspond to looping with an explicit iterator: `for` and `forEach` loops in procedural languages. The most common iterator is a list, as is the case in the `sum` function above. For such cases, Daml has the `foldl` function. The `l` stands for left and means the list is processed from the left. There is also a corresponding `foldr` which processes from the right.

```
foldl : (b -> a -> b) -> b -> [a] -> b
```

Let's give the type parameters semantic names. `b` is the state, `a` is an item. `foldl`s first argument is a function which takes a state and an item and returns a new state. That's the equivalent of the inner block of the `forEach`. It then takes a state, which is the initial state, and a list of items, which is the iterator. The result is again a state. The `sum` function above can be translated to Daml almost instantly with those correspondences in mind:

```
sum ints = foldl (+) 0 ints
```

If we wanted to be more verbose, we could replace `(+)` with a lambda `(\result i -> result + i)` which makes the correspondence to `result += i` from the JavaScript clearer.

Almost all loops with explicit iterators can be translated to folds, though we have to take a bit of care with performance when it comes to translating `for` loops:

```
function sumArrs(arr1, arr2) {
  var l = min (arr1.length, arr2.length);
  var result = new int[l];
  for(var i = 0; i < l; i++) {
    result[i] = arr1[i] + arr2[i];
  }
  return result;
}
```

Translating the `for` into a `forEach` is easy if you can get your hands on an array containing values `[0..(l-1)]`. And that's how you do it in Daml, using *ranges*. `[0..(l-1)]` is shorthand for `enumFromTo 0 (l-1)`, which returns the list you'd expect.

Daml also has an operator `(!!) : [a] -> Int -> a` which returns an element in a list. You may now be tempted to write `sumArrs` like this:

```
sumArrs : [Int] -> [Int] -> [Int]
sumArrs arr1 arr2 =
  let l = min (length arr1) (length arr2)
      sumAtI i = (arr1 !! i) + (arr2 !! i)
   in foldl (\state i -> (sumAtI i) :: state) [] [1..(l-1)]
```

Unfortunately, that's not a very good approach. Lists in Daml are linked lists, which makes access using `(!!)` too slow for this kind of iteration. A better approach in Daml is to get rid of the `i` alto-

gether and instead merge the lists first using the `zip` function, and then iterate over the zipped up lists:

```
sumArrs2 arr1 arr2 = foldl (\state (x, y) -> (x + y) :: state) [] (zip arr1 arr2)
```

`zip : [a] -> [b] -> [(a, b)]` takes two lists, and merges them into a single list where the first element is the 2-tuple containing the first element of the two input lists, and so on. It drops any left-over elements of the longer list, thus making the `min` logic unnecessary.

Maps

In effect, the lambda passed to `foldl` only wants to act on a single element of the (zipped-up) input list, but still has to manage the concatenation of the whole state. Acting on each element separately is a common-enough pattern that there is a specialized function for it: `map : (a -> b) -> [a] -> [b]`. Using it, we can rewrite `sumArr` to:

```
sumArrs3 arr1 arr2 = map (\(x, y) -> (x + y)) (zip arr1 arr2)
```

As a rule, use `map` if the result has the same shape as the input and you don't need to carry state from one iteration to the next. Use folds if you need to accumulate state in any way.

Recursion

If there is no explicit iterator, you can use recursion. Let's try to write a function that reverses a list, for example. We want to avoid `(!!)` so there is no sensible iterator here. Instead, we use recursion:

```
reverseWorker rev rem = case rem of
  [] -> rev
  x::xs -> reverseWorker (x::rev) xs
reverse xs = reverseWorker [] xs
```

You may be tempted to make `reverseWorker` a local definition inside `reverse`, but Daml only supports recursion for top-level functions so the recursive part `recurseWorker` has to be its own top-level function.

Folds and Maps in Action Contexts

The folds and `map` function above are pure in the sense introduced in *Add Constraints to a Contract*: The functions used to map or process items have no side effects. If you have looked at the *Work with Dependencies* models, you'll have noticed `mapA`, `mapA_`, and `forA`, which seem to serve a similar role but within `Action`s . A good example is the `mapA` call in the `testMultiTrade` script:

```
  let rels =
        [ Relationship chfbank alice
        , Relationship chfbank bob
        , Relationship gbpbank alice
        , Relationship gbpbank bob
        ]
  [chfha, chfhb, gbpha, gbphb] <- mapA setupRelationship rels
```

Here we have a list of relationships (type `[Relationship]`) and a function `setupRelationship : Relationship -> Script (ContractId AssetHolder)`. We want the `AssetHolder` contracts for those relationships, i.e. something of type `[ContractId AssetHolder]`. Using the map function almost gets us there, but `map setupRelationship rels` would have type `[Update (ContractId AssetHolder)]`. This is a list of `Update` actions, each resulting in a `ContractId AssetHolder`. What we need is an `Update` action resulting in a `[ContractId AssetHolder]`. The list and `Update` are nested the wrong way around for our purposes.

Intuitively, it's clear how to fix this: we want the compound action consisting of performing each of the actions in the list in turn. There's a function for that: `sequence : : Applicative m => [m a] -> m [a]`. It implements that intuition and allows us to take the `Update` out of the list , so to speak. So we could write `sequence (map setupRelationship rels)`. This is so common that it's encapsulated in the `mapA` function, a possible implementation of which is

```
mapA f xs = sequence (map f xs)
```

The `A` in `mapA` stands for Action , and you'll find that many functions that have something to do with looping have an `A` equivalent. The most fundamental of all of these is `foldlA : Action m => (b -> a -> m b) -> b -> [a] -> m b`, a left fold with side effects. Here the inner function has a side-effect indicated by the `m` so the end result `m b` also has a side effect: the sum of all the side effects of the inner function.

To improve your familiarity with these concepts, try implementing `foldlA` in terms of `foldl`, as well as `sequence` and `mapA` in terms of `foldlA`. Here is one set of possible implementations:

```
foldlA2 fn init xs =
  let
    work accA x = do
      acc <- accA
      fn acc x
   in foldl work (pure init) xs

mapA2 fn xs =
  let
    work ys x = do
      y <- fn x
      return (y :: ys)
   in foldlA2 work [] xs

sequence2 actions =
  let
    work ys action = do
      y <- action
      return (y :: ys)
   in foldlA2 work [] actions
```

`forA` is just `mapA` with its arguments reversed. This is useful for readability if the list of items is already in a variable, but the function is a lengthy lambda.

```
  [usdCid, chfCid] <- forA [usdCid, chfCid] (\cid -> submit alice do
    exerciseCmd cid SetObservers with
      newObservers = [bob]
    )
```

Lastly, you'll have noticed that in some cases we used `mapA_`, not `mapA`. The underscore indicates that the result is not used, so `mapA_ fn xs fn == void (mapA fn xs)`. The Daml Linter will

alert you if you could use `mapA_` instead of `mapA`, and similarly for `forA_`.

1.11.12.4 Next Up

You now know the basics of functions and control flow, both in pure and Action contexts. The *Work with Dependencies* example shows just how much can be done with just the tools you have encountered here, but there are many more tools at your disposal in the Daml Standard Library. It provides functions and typeclasses for many common circumstances and in *Introduction to the Daml Standard Library*, you'll get an overview of the library and learn how to search and browse it.

1.11.13 Introduction to the Daml Standard Library

In *Data Types* and *Functional Programming 101* you learned how to define your own data types and functions. However, you don't have to implement everything from scratch. Daml comes with the *Daml Standard Library*, which contains types, functions, and typeclasses that cover a large range of use cases.

In this chapter, you'll get an overview of the essentials and learn how to browse and search the library to find functions. Being proficient with the Standard Library will make you considerably more efficient writing Daml code. Specifically, this chapter covers:

The Prelude
Important types from the Standard Library, and associated functions and typeclasses
Typeclasses
Important typeclasses like `Functor`, `Foldable`, and `Traversable`
How to search the Standard Library

To go in depth on some of these topics, the literature referenced in *The Haskell Connection* covers them in much greater detail. The Standard Library typeclasses like `Applicative`, `Foldable`, `Traversable`, `Action` (called `Monad` in Haskell), and many more, are the bread and butter of Haskell programmers.

Note: There is a project template `daml-intro-11` for this chapter, but it only contains a single source file with the code snippets embedded in this section.

1.11.13.1 The Prelude

You've already used a lot of functions, types, and typeclasses without importing anything. Functions like `create`, `exercise`, and `(==)`, types like `[]`, `(,)`, `Optional`, and typeclasses like `Eq`, `Show`, and `Ord`. These all come from the *Prelude*. The Prelude is module that gets implicitly imported into every other Daml module and contains both Daml specific machinery as well as the essentials needed to work with the inbuilt types and typeclasses.

1.11.13.2 Important Types From the Prelude

In addition to the *Native Types*, the Prelude defines a number of common types:

Lists

You've already met lists. Lists have two constructors `[]` and `x :: xs`, the latter of which is prepend in the sense that `1 :: [2] == [1, 2]`. In fact `[1,2]` is just syntactical sugar for `1 :: 2 :: []`.

Tuples

In addition to the 2-tuple you have already seen, the Prelude contains definitions for tuples of size up to 15. Tuples allow you to store mixed data in an ad-hoc fashion. Common use-cases are return values from functions consisting of several pieces or passing around data in folds, as you saw in *Folds*. An example of a relatively wide Tuple can be found in the test modules of the *Exception Handling* project. `Test.Intro.Asset.TradeSetup.tradeSetup` returns the allocated parties and active contracts in a long tuple. `Test.Intro.Asset.MultiTrade.testMultiTrade` puts them back into scope using pattern matching:

```
  return (alice, bob, usdbank, eurbank, usdha, usdhb, eurha, eurhb, usdCid,
↪eurCid)
```

```
  (alice, bob, usdbank, eurbank, usdha, usdhb, eurha, eurhb, usdCid, eurCid) <-
↪tradeSetup
```

Tuples, like lists have some syntactic magic. Both the types as well as the constructors for tuples are `(,,,)` where the number of commas determines the arity of the tuple. Type and data constructor can be applied with values inside the brackets, or outside, and partial application is possible:

```
t1 : (Int, Text) = (1, "a")
t2 : (,) Int Text = (1, "a")
t3 : (Int, Text) = (1,) "a"
t4 : a -> (a, Text) = (,"a")
```

Note: While tuples of great lengths are available, it is often advisable to define custom records with named fields for complex structures or long-lived values. Overuse of tuples can harm code readability.

Optional

The `Optional` type represents a value that may be missing. It's the closest thing Daml has to a nullable value. `Optional` has two constructors: `Some`, which takes a value, and `None`, which doesn't take a value. In many languages one would write code like this:

```
lookupResult = lookupByKey(k);

if( lookupResult == null) {
  // Do something
} else {
  // Do something else
}
```

In Daml the same thing would be expressed as:

```
  lookupResult <- lookupByKey @T k
  case lookupResult of
    None -> do -- Do Something
      return ()
    Some cid -> do -- Do Something
      return ()
```

Either

`Either` is used in cases where a value should store one of two types. It has two constructors, `Left` and `Right`, each of which take a value of one or the other of the two types. One typical use-case of `Either` is as an extended `Optional` where `Right` takes the role of `Some` and `Left` the role of `None`, but with the ability to store an error value. `Either Text`, for example behaves just like `Optional`, except that values with constructor `Left` have a text associated to them.

Note: As with tuples, it's easy to overuse `Either` and harm readability. Consider writing your own more explicit type instead. For example if you were returning `South a` vs `North b` using your own type over `Either` would make your code clearer.

1.11.13.3 Typeclasses

You've seen typeclasses in use all the way from *Data Types*. It's now time to look under the hood.

Typeclasses are declared using the `class` keyword:

```
class HasQuantity a q where
  getQuantity : a -> q
  setQuantity : q -> a -> a
```

This is akin to an interface declaration of an interface with a getter and setter for a quantity. To *implement* this interface, you need to define instances of this typeclass:

```
data Foo = Foo with
  amount : Decimal
```

(continues on next page)

(continued from previous page)

```
instance HasQuantity Foo Decimal where
  getQuantity foo = foo.amount
  setQuantity amount foo = foo with amount
```

Typeclasses can have constraints like functions. For example: `class Eq a => Ord a` means everything that is orderable can also be compared for equality . And that's almost all there's to it.

1.11.13.4 Important Typeclasses From the Prelude

Eq

The `Eq` typeclass allows values of a type to be compared for (in)-equality. It makes available two function: `==` and `/=`. Most data types from the Standard Library have an instance of `Eq`. As you already learned in *Data Types*, you can let the compiler automatically derive instances of `Eq` for you using the `deriving` keyword.

Templates always have an `Eq` instance, and all types stored on a template need to have one.

Ord

The `Ord` typeclass allows values of a type to be compared for order. It makes available functions: `<`, `>`, `<=`, and `>=`. Most of the inbuilt data types have an instance of `Ord`. Furthermore, types like `List` and `Optional` get an instance of `Ord` if the type they contain has one. You can let the compiler automatically derive instances of `Ord` for you using the `deriving` keyword.

Show

`Show` indicates that a type can be serialized to `Text`, ie shown in a shell. Its key function is `show`, which takes a value and converts it to `Text`. All inbuilt data types have an instance for `Show` and types like `List` and `Optional` get an instance if the type they contain has one. It also supports the `deriving` keyword.

Functor

Functors are the closest thing to containers that Daml has. Whenever you see a type with a single type parameter, you are probably looking at a `Functor`: `[a]`, `Optional a`, `Either Text a`, `Update a`. Functors are things that can be mapped over and as such, the key function of `Functor` is `fmap`, which does generically what the `map` function does for lists.

Other classic examples of Functors are Sets, Maps, Trees, etc.

Applicative Functor

Applicative Functors are a bit like Actions, which you met in *Add Constraints to a Contract*, except that you can't use the result of one action as the input to another action. The only important Applicative Functor that isn't an action in Daml is the `Commands` type submitted in a `submit` block in Daml Script. That's why in order to use `do` notation in Daml Script, you have to enable the `ApplicativeDo` language extension.

Actions

Actions were already covered in *Add Constraints to a Contract*. One way to think of them is as recipes for a value, which need to be executed to get at that value. Actions are always Functors (and Applicative Functors). The intuition for that is simply that `fmap f x` is the recipe in `x` with the extra instruction to apply the pure function `f` to the result.

The really important Actions in Daml are `Update` and `Script`, but there are many others, like `[]`, `Optional`, and `Either a`.

Semigroups and Monoids

Semigroups and monoids are about binary operations, but in practice, their important use is for `Text` and `[]`, where they allow concatenation using the `{<>}` operator.

Additive and Multiplicative

Additive and Multiplicative abstract out arithmetic operations, so that `(+)`, `(-)`, `(*)`, and some other functions can be used uniformly between `Decimal` and `Int`.

1.11.13.5 Important Modules in the Standard Library

For almost all the types and typeclasses presented above, the Standard Library contains a module:

DA.List for Lists
DA.Optional for `Optional`
DA.Tuple for Tuples
DA.Either for `Either`
DA.Functor for Functors
DA.Action for Actions
DA.Monoid and *DA.Semigroup* for Monoids and Semigroups
DA.Text for working with `Text`
DA.Time for working with `Time`
DA.Date for working with `Date`

You get the idea, the names are fairly descriptive.

Other than the typeclasses defined in Prelude, there are two modules generalizing concepts you've already learned, which are worth knowing about: `Foldable` and `Traversable`. In *Looping* you learned all about folds and their Action equivalents. All the examples there were based on lists, but there are many other possible iterators. This is expressed in two additional typeclasses: *DA.Traversable*, and

DA.Foldable. For more detail on these concepts, please refer to the literature in *The Haskell Connection*, or https://wiki.haskell.org/Foldable_and_Traversable.

1.11.13.6 Search the Standard Library

Being able to browse the Standard Library starting from *The standard library* is a start, and the module naming helps, but it's not an efficient process for finding out what a function you've encountered does, even less so for finding a function that does a thing you need to do.

Daml has its own version of the Hoogle search engine, which offers search both by name and by signature. This function is fully integrated into the search bar on https://docs.daml.com/, but for those wanting a pure Standard Library search, it's also available on https://hoogle.daml.com.

Search for Functions by Name

Say you come across some functions you haven't seen before, like the ones in the `ensure` clause of the `MultiTrade`.

```
    ensure (length baseAssetCids == length baseAssets) &&
      (length quoteApprovalCids == length quoteAssets) &&
      not (null baseAssets) &&
      not (null quoteAssets)
```

You may be able to guess what `not` and `null` do, but try searching those names in the documentation search. Search results from the Standard Library will show on top. `not`, for example, gives

```
not

 : Bool -> Bool

 Boolean "not"
```

Signature (including type constraints) and description usually give a pretty clear picture of what a function does.

Search for Functions by Signature

The other very common use case for the search is that you have some values that you want to do something with, but don't know the standard library function you need. On the `MultiTrade` template we have a list `baseAssets`, and thanks to your ensure clause we know it's non-empty. In the original `Trade` we used `baseAsset.owner` as the signatory. How do you get the first element of this list to extract the `owner` without going through the motions of a complete pattern match using `case`?

The trick is to think about the signature of the function that's needed, and then to search for that signature. In this case, we want a single distinguished element from a list so the signature should be `[a] -> a`. If you search for that, you'll get a whole range of results, but again, Standard Library results are shown at the top.

Scanning the descriptions, `head` is the obvious choice, as used in the `let` of the `MultiTrade` template.

You may notice that in the search results you also get some hits that don't mention `[]` explicitly. For example:

The reason is that there is an instance for `Foldable [a]`.

Let's try another search. Suppose you didn't want the first element, but the one at index `n`. Remember that `(!!)` operator from *Functional Programming 101*? There are now two possible signatures we could search for: `[a] -> Int -> a` and `Int -> [a] -> a`. Try searching for both. You'll see that the search returns `(!!)` in both cases. You don't have to worry about the order of arguments.

1.11.13.7 Next Up

In the following section, you'll find options for testing and interacting with Daml code. We also talk about the operational semantics of some keywords and their commonly associated failures, and a little bit about how coverage reports work in Daml testing.

1.11.14 Good Design Patterns

Patterns have been useful in the programming world, as both a source of design inspiration, and a document of good design practices. This document is a catalog of Daml patterns intended to provide the same facility in the Daml application world.

You can checkout the examples locally via `daml new daml-patterns --template daml-patterns`.

The Initiate and Accept Pattern The Initiate and Accept pattern demonstrates how to start a bilateral workflow. One party initiates by creating a proposal or an invite contract. This gives another party the chance to accept, reject or renegotiate.

The Multiple Party Agreement Pattern The Multiple Party Agreement pattern uses a Pending contract as a wrapper for the Agreement contract. Any one of the signatory parties can kick off the workflow by creating a Pending contract on the ledger, filling in themselves in all the signatory fields. The Agreement contract is not created on the ledger until all parties have agreed to the Pending contract, and replaced the initiator's signature with their own.

The Delegation Pattern The Delegation pattern gives one party the right to exercise a choice on behalf of another party. The agent can control a contract on the ledger without the principal explicitly committing the action.

The Authorization Pattern The Authorization pattern demonstrates how to make sure a controlling party is authorized before they take certain actions.

The Locking Pattern The Locking pattern exhibits how to achieve locking safely and efficiently in Daml. Only the specified locking party can lock the asset through an active and authorized action. When a contract is locked, some or all choices specified on that contract may not be exercised.

1.11.14.1 The Initiate and Accept Pattern

The Initiate and Accept pattern demonstrates how to start a bilateral workflow. One party initiates by creating a proposal or an invite contract. This gives another party the chance to accept, reject or renegotiate.

Motivation

It takes two to tango, but one party has to initiate. It is no different in the business world. The contractual relationship between two businesses often starts with an invite, a business proposal, a bid offering, etc.

Invite When a market operator wants to set up a market, they need to go through an onboarding process in which they invite participants to sign master service agreements and fulfill different roles in the market. Receiving participants need to evaluate the rights and responsibilities of each role and respond accordingly.

Propose When issuing an asset, an issuer is making a business proposal to potential buyers. The proposal lays out what is expected from buyers, and what they can expect from the issuer. Buyers need to evaluate all aspects of the offering, e.g. price, return, and tax implications, before making a decision.

The Initiate and Accept pattern demonstrates how to write a Daml program to model the initiation of an inter-company contractual relationship. Daml modelers often have to follow this pattern to ensure that no participant is forced into an obligation.

Implementation

The Initiate and Accept pattern in general involves two contracts, the initiate contract and the result contract:

Initiate Contract The initiate contract can be created from a role contract or any other point in the workflow. In this example, the initiate contract is the proposal contract *CoinIssueProposal* which the issuer created from the master contract *CoinMaster*.

```
template CoinMaster
  with
    issuer: Party
  where
    signatory issuer

    nonconsuming choice Invite : ContractId CoinIssueProposal
      with owner: Party
      controller issuer
      do create CoinIssueProposal
            with coinAgreement = CoinIssueAgreement with issuer; owner
```

The *CoinIssueProposal* contract has *Issuer* as the signatory and *Owner* as the controller to the *Accept* choice. In its complete form, the *CoinIssueProposal* contract should define all choices available to the owner, i.e. Accept, Reject or Counter (re-negotiate terms).

```
template CoinIssueProposal
  with
```

(continues on next page)

(continued from previous page)

```
    coinAgreement: CoinIssueAgreement
  where
    signatory coinAgreement.issuer
    observer coinAgreement.owner

    choice AcceptCoinProposal
      : ContractId CoinIssueAgreement
      controller coinAgreement.owner
      do create coinAgreement
```

Result Contract Once the owner exercises the *AcceptCoinProposal* choice on the initiate contract to express their consent, it returns a result contract representing the agreement between the two parties. In this example, the result contract is of type *CoinIssueAgreement*. Note, it has both *issuer* and *owner* as the signatories, implying they both need to consent to the creation of this contract. Both parties could be controller(s) on the result contract, depending on the business case.

```
template CoinIssueAgreement
  with
    issuer: Party
    owner: Party
  where
    signatory issuer, owner

    nonconsuming choice Issue : ContractId Coin
      with amount: Decimal
      controller issuer
      do create Coin with issuer; owner; amount; delegates = []
```

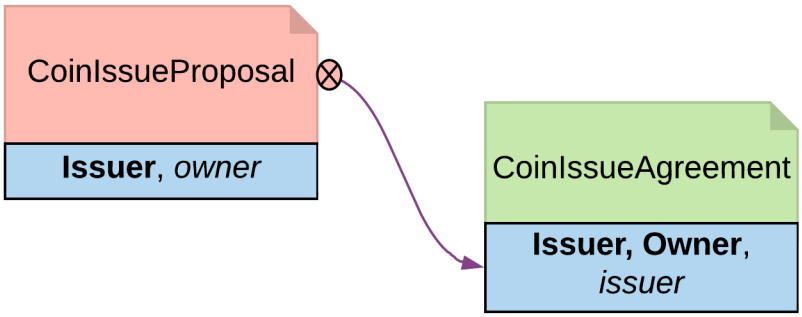


Fig. 1: Initiate and Accept pattern diagram

Trade-offs

Initiate and Accept can be quite verbose if signatures from more than two parties are required to progress the workflow.

1.11.14.2 The Multiple Party Agreement Pattern

The Multiple Party Agreement pattern uses a Pending contract as a wrapper for the Agreement contract. Any one of the signatory parties can kick off the workflow by creating a Pending contract on the ledger, filling in themselves in all the signatory fields. The Agreement contract is not created on the ledger until all parties have agreed to the Pending contract, and replaced the initiator's signature with their own.

Motivation

The *The Initiate and Accept Pattern* shows how to create bilateral agreements in Daml. However, a project or a workflow often requires more than two parties to reach a consensus and put their signatures on a multi-party contract. For example, in a large construction project, there are at least three major stakeholders: Owner, Architect and Builder. All three parties need to establish agreement on key responsibilities and project success criteria before starting the construction.

If such an agreement were modeled as three separate bilateral agreements, no party could be sure if there are conflicts between their two contracts and the third contract between their partners. If the *The Initiate and Accept Pattern* were used to collect three signatures on a multi-party agreement, unnecessary restrictions would be put on the order of consensus and a number of additional contract templates would be needed as the intermediate steps. Both solution are suboptimal.

Following the Multiple Party Agreement pattern, it is easy to write an agreement contract with multiple signatories and have each party accept explicitly.

Implementation

Agreement contract The *Agreement* contract represents the final agreement among a group of stakeholders. Its content can vary per business case, but in this pattern, it always has multiple signatories.

```
template Agreement
  with
    signatories: [Party]
  where
    signatory signatories
    ensure
      unique signatories
  -- The rest of the template to be agreed to would follow here
```

Pending contract The *Pending* contract needs to contain the contents of the proposed *Agreement* contract, as a parameter. This is so that parties know what they are agreeing to, and also so that when all parties have signed, the *Agreement* contract can be created.
The *Pending* contract has a list of parties who have signed it, and a list of parties who have yet to sign it. If you add these lists together, it has to be the same set of parties as the `signatories` of the *Agreement* contract.

All of the `toSign` parties have the choice to `Sign`. This choice checks that the party is indeed a member of `toSign`, then creates a new instance of the *Pending* contract where they have been moved to the `signed` list.

```
template Pending
  with
    finalContract: Agreement
    alreadySigned: [Party]
  where
    signatory alreadySigned
    observer finalContract.signatories
    ensure
      -- Can't have duplicate signatories
      unique alreadySigned

    -- The parties who need to sign is the finalContract.signatories with
↪alreadySigned filtered out
    let toSign = filter (`notElem` alreadySigned) finalContract.signatories

    choice Sign : ContractId Pending with
        signer : Party
      controller signer
        do
          -- Check the controller is in the toSign list, and if they are,
↪sign the Pending contract
          assert (signer `elem` toSign)
          create this with alreadySigned = signer :: alreadySigned
```

Once all of the parties have signed, any of them can create the final Agreement contract using the `Finalize` choice. This checks that all of the signatories for the *Agreement* have signed the *Pending* contract.

```
    choice Finalize : ContractId Agreement with
        signer : Party
      controller signer
        do
          -- Check that all the required signatories have signed Pending
          assert (sort alreadySigned == sort finalContract.signatories)
          create finalContract
```

Collecting the signatures in practice Since the final Pending contract has multiple signatories, **it cannot be created in that state by any one stakeholder**.
However, a party can create a pending contract, with all of the other parties in the `toSign` list.

```
  parties@[person1, person2, person3, person4] <- makePartiesFrom ["Alice",
↪"Bob", "Clare", "Dave"]
  let finalContract = Agreement with signatories = parties

  -- Parties cannot create a contract already signed by someone else
  initialFailTest <- person1 `submitMustFail` do
    createCmd Pending with finalContract; alreadySigned = [person1, person2]

  -- Any party can create a Pending contract provided they list themselves as
↪the only signatory
  pending <- person1 `submit` do
    createCmd Pending with finalContract; alreadySigned = [person1]
```

Once the Pending contract is created, the other parties can sign it. For simplicity, the example code only has choices to express consensus (but you might want to add choices to Accept,

Reject, or Negotiate).

```
-- Each signatory of the finalContract can Sign the Pending contract
pending <- person2 `submit` do
  exerciseCmd pending Sign with signer = person2
pending <- person3 `submit` do
  exerciseCmd pending Sign with signer = person3
pending <- person4 `submit` do
  exerciseCmd pending Sign with signer = person4

-- A party can't sign the Pending contract twice
pendingFailTest <- person3 `submitMustFail` do
  exerciseCmd pending Sign with signer = person3
-- A party can't sign on behalf of someone else
pendingFailTest <- person3 `submitMustFail` do
  exerciseCmd pending Sign with signer = person4
```

Once all of the parties have signed the Pending contract, any of them can then exercise the `Finalize` choice. This creates the Agreement contract on the ledger.

```
person1 `submit` do
  exerciseCmd pending Finalize with signer = person1
```

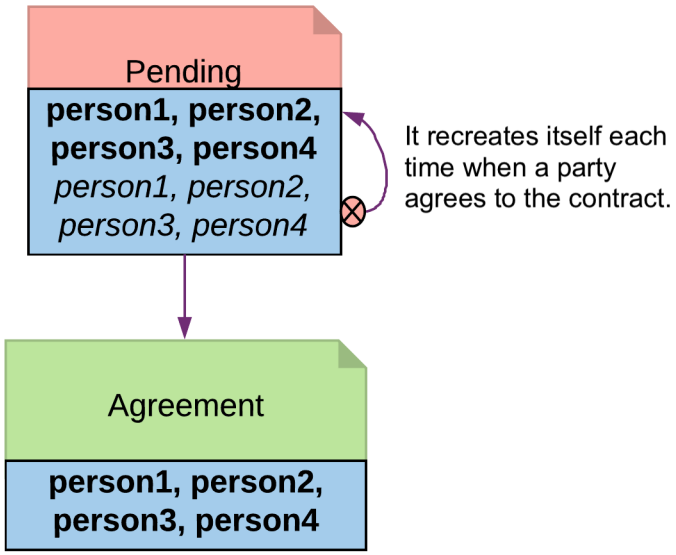


Fig. 2: Multiple Party Agreement Diagram

1.11.14.3 The Delegation Pattern

The Delegation pattern gives one party the right to exercise a choice on behalf of another party. The agent can control a contract on the ledger without the principal explicitly committing the action.

Motivation

Delegation is prevalent in the business world. In fact, the entire custodian business is based on delegation. When a company chooses a custodian bank, it is effectively giving the bank the rights to hold their securities and settle transactions on their behalf. The securities are not legally possessed by the custodian banks, but the banks should have full rights to perform actions in the client's name, such as making payments or changing investments.

The Delegation pattern enables Daml modelers to model the real-world business contractual agreements between custodian banks and their customers. Ownership and administration rights can be segregated easily and clearly.

Implementation

Pre-condition: There exists a contract, on which controller Party A has a choice and intends to delegate execution of the choice to Party B. In this example, the owner of a *Coin* contract intends to delegate the *Transfer* choice.

```
template Coin
  with
    owner: Party
    issuer: Party
    amount: Decimal
    delegates : [Party]
  where
    signatory issuer, owner
    observer delegates
```

```
    choice Transfer : ContractId TransferProposal
      with newOwner: Party
      controller owner
      do
          create TransferProposal
            with coin=this; newOwner
```

Delegation Contract

Principal, the original coin owner, is the signatory of delegation contract *CoinPoA*. This signatory is required to authorize the *Transfer* choice on *coin*.

```
template CoinPoA
  with
    attorney: Party
    principal: Party
  where
    signatory principal
    observer attorney
```

(continues on next page)

(continued from previous page)

```
choice WithdrawPoA
  : ()
  controller principal
  do return ()
```

Whether or not the *Attorney* party should be a signatory of *CoinPoA* is subject to the business agreements between *Principal* and *Attorney*. For simplicity, in this example, *Attorney* is not a signatory.
Attorney is the controller of the Delegation choice on the contract. Within the choice, *Principal* exercises the choice *Transfer* on the Coin contract.

```
nonconsuming choice TransferCoin
  : ContractId TransferProposal
  with
    coinId: ContractId Coin
    newOwner: Party
  controller attorney
  do
    exercise coinId Transfer with newOwner
```

Coin contracts need to be disclosed to *Attorney* before they can be used in an exercise of *Transfer*. This can be done by adding *Attorney* to *Coin* as an Observer. This can be done dynamically, for any specific *Coin*, by making the observers a *List*, and adding a choice to add a party to that List:

```
choice Disclose : ContractId Coin
  with p : Party
  controller owner
  do create this with delegates = p :: delegates
```

Note: The technique is likely to change in the future. Daml is actively researching future language features for contract disclosure.

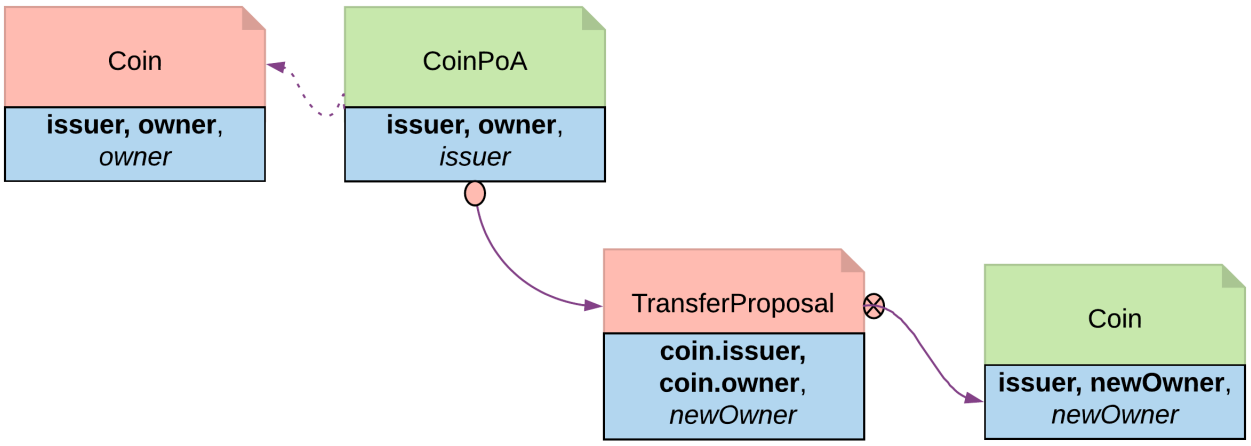


Fig. 3: Delegation pattern diagram

1.11.14.4 The Authorization Pattern

The Authorization pattern demonstrates how to make sure a controlling party is authorized before they take certain actions.

Motivation

Authorization is an universal concept in the business world as access to most business resources is a privilege, and not given freely. For example, security trading may seem to be a plain bilateral agreement between the two trading counterparties, but this could not be further from truth. To be able to trade, the trading parties need go through a series of authorization processes and gain permission from a list of service providers such as exchanges, market data streaming services, clearing houses and security registrars etc.

The Authorization pattern shows how to model these authorization checks prior to a business transaction.

Authorization

Here is an implementation of a *Coin transfer* without any authorization:

```
template Coin
  with
    owner: Party
    issuer: Party
    amount: Decimal
    delegates : [Party]
  where
    signatory issuer, owner
    observer delegates
```

```
    choice Transfer : ContractId TransferProposal
      with newOwner: Party
      controller owner
      do
          create TransferProposal
            with coin=this; newOwner
```

This is may be insufficient since the issuer has no means to ensure the newOwner is an accredited company. The following changes fix this deficiency.

Authorization contract The below shows an authorization contract *CoinOwnerAuthorization*. In this example, the issuer is the only signatory so it can be easily created on the ledger. Owner is an observer on the contract to ensure they can see and use the authorization.

```
template CoinOwnerAuthorization
  with
    owner: Party
    issuer: Party
  where
    signatory issuer
    observer owner
```

(continues on next page)

(continued from previous page)

```
    choice WithdrawAuthorization
      : ()
      controller issuer
      do return ()
```

Authorization contracts can have much more advanced business logic, but in its simplest form, *CoinOwnerAuthorization* serves its main purpose, which is to prove the owner is a warranted coin owner.

TransferProposal contract In the TransferProposal contract, the Accept choice checks that newOwner has proper authorization. A *CoinOwnerAuthorization* for the new owner has to be supplied and is checked by the two assert statements in the choice before a coin can be transferred.

```
    choice AcceptTransfer
      : ContractId Coin
      with token: ContractId CoinOwnerAuthorization
      controller newOwner
      do
        t <- fetch token
        assert (coin.issuer == t.issuer)
        assert (newOwner == t.owner)
        create coin with owner = newOwner
```

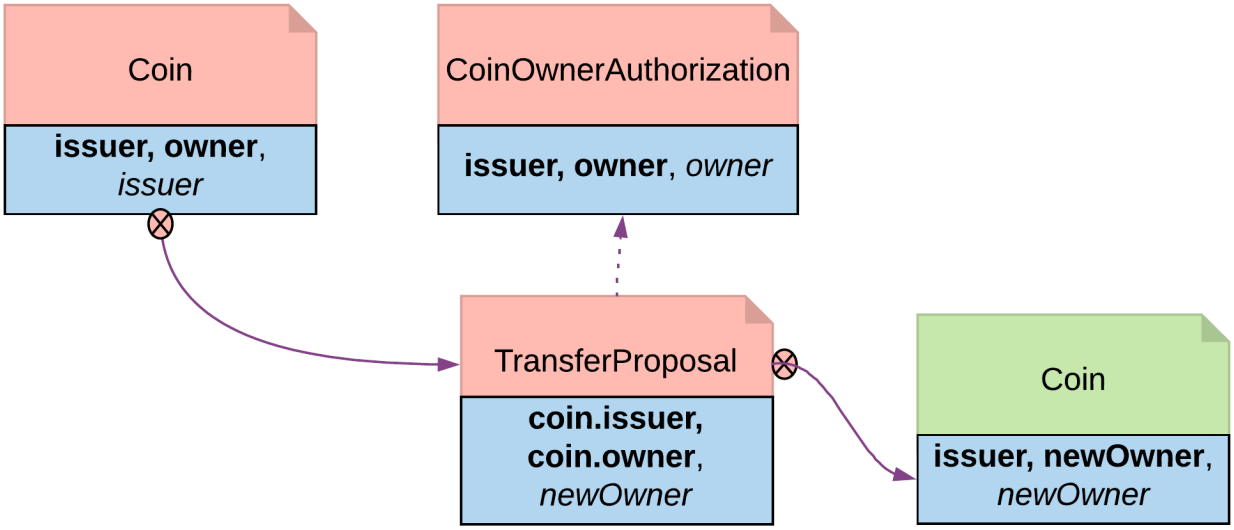


Fig. 4: Authorization Diagram

1.11.14.5 The Locking Pattern

The Locking pattern exhibits how to achieve locking safely and efficiently in Daml. Only the specified locking party can lock the asset through an active and authorized action. When a contract is locked, some or all choices specified on that contract may not be exercised.

Motivation

Locking is a common real-life requirement in business transactions. During the clearing and settlement process, once a trade is registered and novated to a central Clearing House, the trade is considered locked-in. This means the securities under the ownership of seller need to be locked so they cannot be used for other purposes, and so should be the funds on the buyer's account. The locked state should remain throughout the settlement Payment versus Delivery process. Once the ownership is exchanged, the lock is lifted for the new owner to have full access.

Implementation

There are three ways to achieve locking:

Lock by Archiving

Pre-condition: there exists a contract that needs to be locked and unlocked. In this section, *Coin* is used as the original contract to demonstrate locking and unlocking.

```
template Coin
  with
    owner: Party
    issuer: Party
    amount: Decimal
    delegates : [Party]
  where
    signatory issuer, owner
    observer delegates
```

```
    choice Transfer : ContractId TransferProposal
      with newOwner: Party
      controller owner
      do
          create TransferProposal
            with coin=this; newOwner
```

```
    --a coin can only be archived by the issuer under the condition that the
↪issuer is the owner of the coin. This ensures the issuer cannot archive coins
↪at will.
    choice Archives
      : ()
      controller issuer
      do assert (issuer == owner)
```

Archiving is a straightforward choice for locking because once a contract is archived, all choices on the contract become unavailable. Archiving can be done either through consuming choice or archiving contract.

Consuming Choice

The steps below show how to use a consuming choice in the original contract to achieve locking:

Add a consuming choice, *Lock*, to the *Coin* template that creates a *LockedCoin*.
The controller party on the *Lock* may vary depending on business context. In this example, *owner* is a good choice.
The parameters to this choice are also subject to business use case. Normally, it should have at least locking terms (eg. lock expiry time) and a party authorized to unlock.

```
choice Lock : ContractId LockedCoin
  with maturity: Time; locker: Party
  controller owner
  do create LockedCoin with coin=this; maturity; locker
```

Create a *LockedCoin* to represent *Coin* in the locked state. *LockedCoin* has the following characteristics, all in order to be able to recreate the original *Coin*:

- The signatories are the same as the original contract.
- It has all data of *Coin*, either through having a *Coin* as a field, or by replicating all data of *Coin*.
- It has an *Unlock* choice to lift the lock.

```
template LockedCoin
  with
    coin: Coin
    maturity: Time
    locker: Party
  where
    signatory coin.issuer, coin.owner
    observer locker
```

```
    choice Unlock
      : ContractId Coin
      controller locker
      do create coin
```

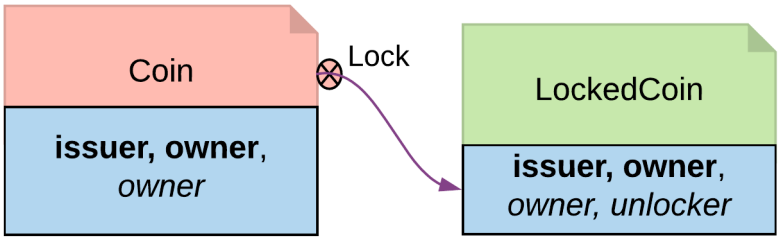


Fig. 5: Locking By Consuming Choice Diagram

Archiving Contract

In the event that changing the original contract is not desirable and assuming the original contract already has an *Archive* choice, you can introduce another contract, *CoinCommitment*, to archive *Coin* and create *LockedCoin*.

Examine the controller party and archiving logic in the *Archives* choice on the *Coin* contract. A coin can only be archived by the issuer under the condition that the issuer is the owner of the coin. This ensures the issuer cannot archive any coin at will.

```
    --a coin can only be archived by the issuer under the condition that the
↪issuer is the owner of the coin. This ensures the issuer cannot archive coins
↪at will.
    choice Archives
      : ()
      controller issuer
      do assert (issuer == owner)
```

Since we need to call the *Archives* choice from *CoinCommitment*, its signatory has to be *Issuer*.

```
template CoinCommitment
  with
    owner: Party
    issuer: Party
    amount: Decimal
   where
    signatory issuer
    observer owner
```

The controller party and parameters on the *Lock* choice are the same as described in locking by consuming choice. The additional logic required is to transfer the asset to the issuer, and then explicitly call the *Archive* choice on the *Coin* contract.
Once a *Coin* is archived, the *Lock* choice creates a *LockedCoin* that represents *Coin* in locked state.

```
    nonconsuming choice LockCoin
      : ContractId LockedCoin
      with coinCid: ContractId Coin
           maturity: Time
           locker: Party
      controller owner
      do
        inputCoin <- fetch coinCid
        assert (inputCoin.owner == owner && inputCoin.issuer == issuer &&
↪inputCoin.amount == amount )
        --the original coin firstly transferred to issuer and then archived
        prop <- exercise coinCid Transfer with newOwner = issuer
        do
          id <- exercise prop AcceptTransfer
          exercise id Archives
        --create a lockedCoin to represent the coin in locked state
        create LockedCoin with
          coin=inputCoin with owner; issuer; amount
          maturity; locker
```

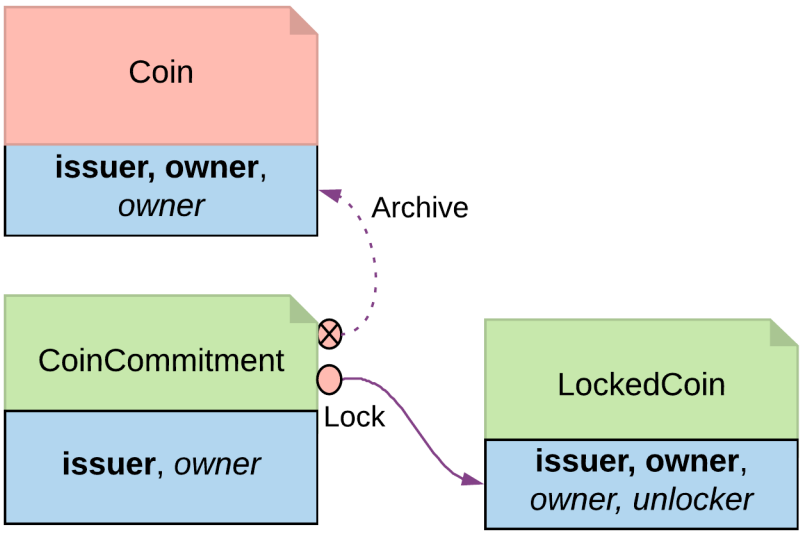


Fig. 6: Locking By Archiving Contract Diagram

Trade-offs

This pattern achieves locking in a fairly straightforward way. However, there are some tradeoffs.

> Locking by archiving disables all choices on the original contract. Usually for consuming choices this is exactly what is required. But if a party needs to selectively lock only some choices, remaining active choices need to be replicated on the *LockedCoin* contract, which can lead to code duplication.
> The choices on the original contract need to be altered for the lock choice to be added. If this contract is shared across multiple participants, it will require agreement from all involved.

Lock by State

The original *Coin* template is shown below. This is the basis on which to implement locking by state

```
template Coin
  with
    owner: Party
    issuer: Party
    amount: Decimal
    delegates : [Party]
  where
    signatory issuer, owner
    observer delegates
```

```
    choice Transfer : ContractId TransferProposal
      with newOwner: Party
      controller owner
      do
          create TransferProposal
            with coin=this; newOwner
```

```
    --a coin can only be archived by the issuer under the condition that the
↪issuer is the owner of the coin. This ensures the issuer cannot archive coins
↪at will.
    choice Archives
      : ()
      controller issuer
      do assert (issuer == owner)
```

In its original form, all choices are actionable as long as the contract is active. Locking by State requires introducing fields to track state. This allows for the creation of an active contract in two possible states: locked or unlocked. A Daml modeler can selectively make certain choices actionable only if the contract is in unlocked state. This effectively makes the asset lockable.

The state can be stored in many ways. This example demonstrates how to create a *LockableCoin* through a party. Alternatively, you can add a lock contract to the asset contract, use a boolean flag or include lock activation and expiry terms as part of the template parameters.

Here are the changes we made to the original *Coin* contract to make it lockable.

Add a *locker* party to the template parameters.
Define the states.
- if owner == locker, the coin is unlocked
- if owner != locker, the coin is in a locked state

The contract state is checked on choices.
- *Transfer* choice is only actionable if the coin is unlocked
- *Lock* choice is only actionable if the coin is unlocked and a 3rd party locker is supplied
- *Unlock* is available to the locker party only if the coin is locked

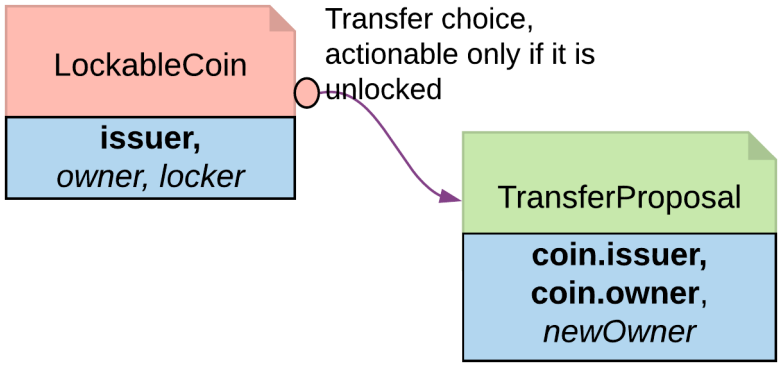
```
template LockableCoin
  with
    owner: Party
    issuer: Party
    amount: Decimal
    locker: Party
  where
    signatory issuer
    signatory owner
    observer locker

    ensure amount > 0.0

    --Transfer can happen only if it is not locked
    choice Transfer : ContractId TransferProposal
      with newOwner: Party
      controller owner
      do
        assert (locker == owner)
        create TransferProposal
          with coin=this; newOwner

      --Lock can be done if owner decides to bring a locker on board
    choice Lock : ContractId LockableCoin
      with newLocker: Party
      controller owner
      do
        assert (newLocker /= owner)
```

(continues on next page)

(continued from previous page)

```
        create this with locker = newLocker

    --Unlock only makes sense if the coin is in locked state
    choice Unlock
      : ContractId LockableCoin
      controller locker
      do
        assert (locker /= owner)
        create this with locker = owner
```

Locking By State Diagram



Trade-offs

It requires changes made to the original contract template. Furthermore you should need to change all choices intended to be locked.
If locking and unlocking terms (e.g. lock triggering event, expiry time, etc) need to be added to the template parameters to track the state change, the template can get overloaded.

Lock by Safekeeping

Safekeeping is a realistic way to model locking as it is a common practice in many industries. For example, during a real estate transaction, purchase funds are transferred to the sellers lawyer's escrow account after the contract is signed and before closing. To understand its implementation, review the original *Coin* template first.

```
template Coin
  with
    owner: Party
    issuer: Party
    amount: Decimal
    delegates : [Party]
  where
    signatory issuer, owner
    observer delegates
```

```
    choice Transfer : ContractId TransferProposal
      with newOwner: Party
      controller owner
      do
          create TransferProposal
            with coin=this; newOwner
```

```
    --a coin can only be archived by the issuer under the condition that the
↪issuer is the owner of the coin. This ensures the issuer cannot archive coins
↪at will.
    choice Archives
      : ()
      controller issuer
      do assert (issuer == owner)
```

There is no need to make a change to the original contract. With two additional contracts, we can transfer the *Coin* ownership to a locker party.

Introduce a separate contract template *LockRequest* with the following features:

- LockRequest has a locker party as the single signatory, allowing the locker party to unilaterally initiate the process and specify locking terms.
- Once owner exercises *Accept* on the lock request, the ownership of coin is transferred to the locker.
- The *Accept* choice also creates a *LockedCoinV2* that represents *Coin* in locked state.

```
template LockRequest
  with
    locker: Party
    maturity: Time
    coin: Coin
  where
    signatory locker
    observer coin.owner

    choice Accept : LockResult
      with coinCid : ContractId Coin
      controller coin.owner
      do
        inputCoin <- fetch coinCid
        assert (inputCoin == coin)
        tpCid <- exercise coinCid Transfer with newOwner = locker
        coinCid <- exercise tpCid AcceptTransfer
        lockCid <- create LockedCoinV2 with locker; maturity; coin
        return LockResult {coinCid; lockCid}
```

LockedCoinV2 represents *Coin* in the locked state. It is fairly similar to the *LockedCoin* described in *Consuming Choice*. The additional logic is to transfer ownership from the locker back to the owner when *Unlock* or *Clawback* is called.

```
template LockedCoinV2
  with
    coin: Coin
    maturity: Time
    locker: Party
  where
```

(continues on next page)

(continued from previous page)

```
    signatory locker, coin.owner

    choice UnlockV2
      : ContractId Coin
      with coinCid : ContractId Coin
      controller locker
      do
        inputCoin <- fetch coinCid
        assert (inputCoin.owner == locker)
        tpCid <- exercise coinCid Transfer with newOwner = coin.owner
        exercise tpCid AcceptTransfer

    choice ClawbackV2
      : ContractId Coin
      with coinCid : ContractId Coin
      controller coin.owner
      do
        currTime <- getTime
        assert (currTime >= maturity)
        inputCoin <- fetch coinCid
        assert (inputCoin == coin with owner=locker)
        tpCid <- exercise coinCid Transfer with newOwner = coin.owner
        exercise tpCid AcceptTransfer
```

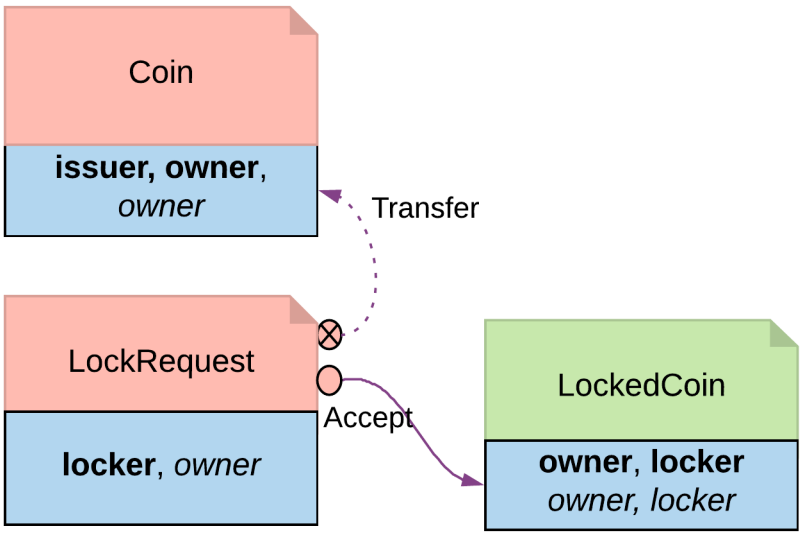
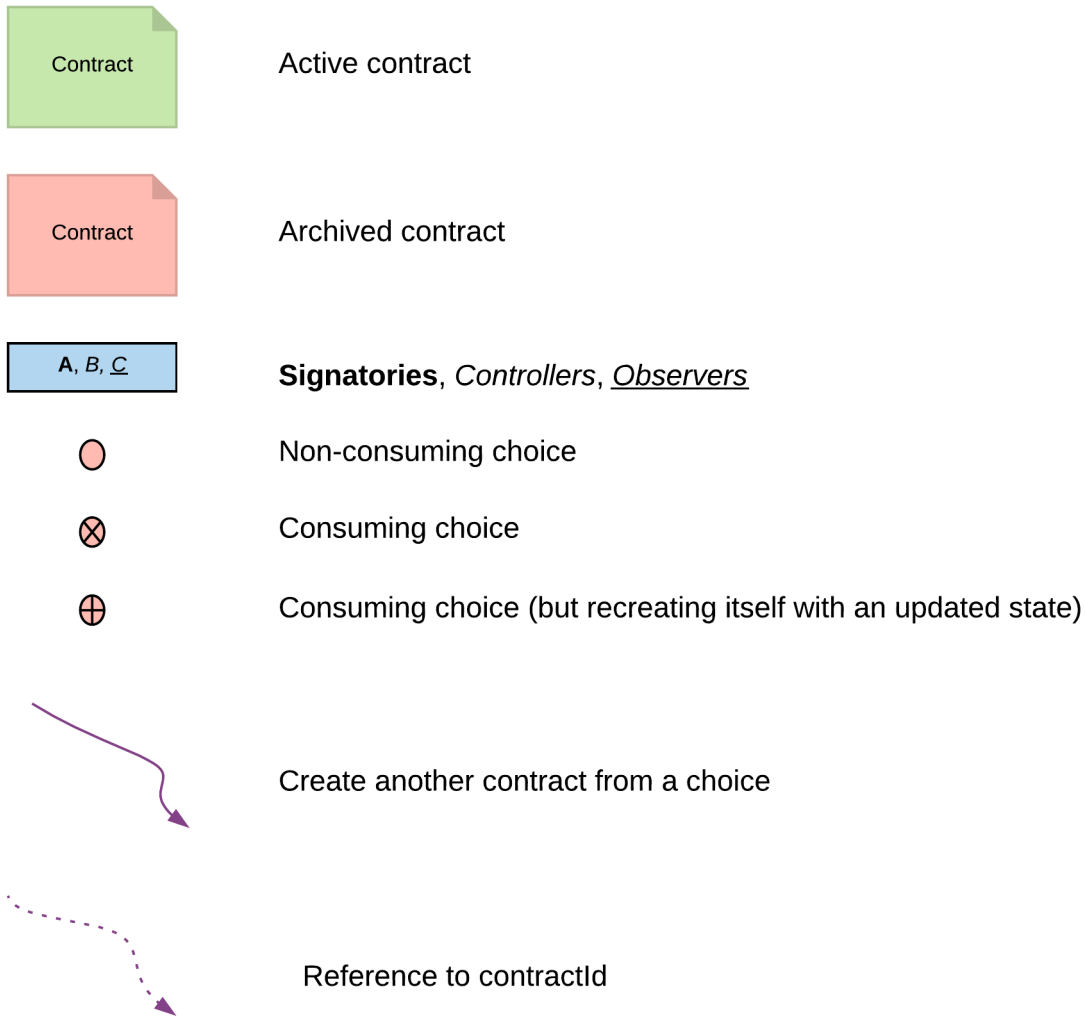


Fig. 7: Locking By Safekeeping Diagram

Trade-offs

Ownership transfer may give the locking party too much access on the locked asset. A rogue lawyer could run away with the funds. In a similar fashion, a malicious locker party could introduce code to transfer assets away while they are under their ownership.

1.11.14.6 Diagram Legends



1.11.15 Test Daml Contracts

This chapter is all about testing and debugging the Daml contracts you've built using the tools from earlier chapters. You've already met Daml Script as a way of testing your code inside the IDE. In this chapter you'll learn about more ways to test with Daml Script and its other uses, as well as other tools you can use for testing and debugging. You'll also learn about a few error cases that are most likely to crop up only in actual distributed testing, and which need some care to avoid. Specifically we will cover:

Daml Test tooling - Script, REPL, and Navigator
Checking choice coverage
The `trace` and `debug` functions
Contention

Note that this section only covers testing your Daml contracts. For more holistic application testing, please refer to *Testing Your Web App*.

If you no longer have your projects set up, load all the code for this parts 1 and 2 of this section into two folders `intro12-part1` and `intro12-part2`, by running `daml new intro12-part1 --template daml-intro-12-part1` and `daml new intro12-part2 --template daml-intro-12-part2`.

1.11.15.1 Daml Test Tooling

There are three primary tools available in the SDK to test and interact with Daml contracts. It is highly recommended to explore the respective docs. The *Work with Dependencies* model lends itself well to being tested using these tools.

Daml Script

Daml Script should be familiar by now. It's a way to script commands and queries from multiple parties against a Daml Ledger. Unless you've browsed other sections of the documentation already, you have probably used it mostly in the IDE. However, Daml Script can do much more than that. It has four different modes of operation:

1. Run on a special Script Service in the IDE, providing the Script Views.
2. Run the Script Service via the CLI, which is useful for quick regression testing.
3. Start a Sandbox and run against that for regression testing against an actual Ledger API.
4. Run against any other already running Ledger.

Daml Navigator

Daml Navigator is a UI that runs against a Ledger API and allows interaction with contracts.

Daml REPL

If you want to do things interactively, Daml REPL is the tool to use. The best way to think of Daml REPL is as an interactive version of Daml Script, but it doubles up as a language REPL (Read-Evaluate-Print Loop), allowing you to evaluate pure expressions and inspect the results.

1.11.15.2 Debug, Trace, and Stacktraces

The above demonstrates nicely how to test the happy path, but what if a function doesn't behave as you expected? Daml has two functions that allow you to do fine-grained printf debugging: `debug` and `trace`. Both allow you to print something to StdOut if the code is reached. The difference between `debug` and `trace` is similar to the relationship between `abort` and `error`:

`debug : Text -> m ()` maps a text to an Action that has the side-effect of printing to StdOut.
`trace : Text -> a -> a` prints to StdOut when the expression is evaluated.

```
daml> let a : Script () = debug "foo"
daml> let b : Script () = trace "bar" (debug "baz")
[Daml.Script:378]: "bar"
daml> a
[DA.Internal.Prelude:532]: "foo"
daml> b
[DA.Internal.Prelude:532]: "baz"
daml>
```

If in doubt, use `debug`. It's the easier of the two to interpret the results of.

The thing in the square brackets is the last location. It'll tell you the Daml file and line number that triggered the printing, but often no more than that because full stacktraces could violate subtransaction privacy quite easily. If you want to enable stacktraces for some purely functional code in your modules, you can use the machinery in *DA.Stack* to do so, but we won't cover that any further here.

1.11.15.3 Diagnose Contention Errors

The above tools and functions allow you to diagnose most problems with Daml code, but they are all synchronous. The sequence of commands is determined by the sequence of inputs. That means one of the main pitfalls of distributed applications doesn't come into play: Contention.

Contention refers to conflicts over access to contracts. Daml guarantees that there can only be one consuming choice exercised per contract so what if two parties simultaneously submit an exercise command on the same contract? Only one can succeed. Contention can also occur due to incomplete or stale knowledge. Maybe a contract was archived a little while ago, but due to latencies, a client hasn't found out yet, or maybe due to the privacy model, they never will. What all these cases have in common is that someone has incomplete knowledge of the state the ledger will be in at the time a transaction will be processed and/or committed.

For in-depth information, see the section on Avoiding Contention.

If we look back at *Daml's Execution Model* we'll see there are three places where ledger state is read:

1. A command is submitted by some client, probably looking at the state of the ledger to build that command. Maybe the command includes references to ContractIds that the client believes are active.
2. During interpretation, ledger state is used to look up active contracts.
3. During commit, ledger state is again used to look up contracts and validate the transaction by reinterpreting it.

Collisions can occur both between 1 and 2 and between 2 and 3. Only during the commit phase is the complete relevant ledger state at the time of the transaction known, which means the ledger state at commit time is king. As a Daml contract developer, you need to understand the different causes

of contention, be able to diagnose the root cause if errors of this type occur, and be able to avoid collisions by designing contracts appropriately.

Common Errors

The most common error messages you'll see are listed below. All of them can be due to one of three reasons.

1. Race Conditions - knowledge of a state change is not yet known during command submission
2. Stale References - the state change is known, but contracts have stale references to keys or ContractIds
3. Ignorance - due to privacy or operational semantics, the requester doesn't know the current state

Following the possible error messages, we'll discuss a few possible causes and remedies.

ContractId Not Found During Interpretation

```
Command interpretation error in LF-Damle: dependency error: couldn't find
↪contract
↪ContractId(004481eb78464f1ed3291b06504d5619db4f110df71cb5764717e1c4d3aa096b9f).
```

ContractId Not Found During Validation

```
Disputed: dependency error: couldn't find contract ContractId
↪(00c06fa370f8858b20fd100423d928b1d200d8e3c9975600b9c038307ed6e25d6f).
```

fetchByKey Error During Interpretation

```
Command interpretation error in LF-Damle: dependency error: couldn't find key com.
↪daml.lf.transaction.GlobalKey@11f4913d.
```

fetchByKey Dispute During Validation

```
Disputed: dependency error: couldn't find key com.daml.lf.transaction.
↪GlobalKey@11f4913d
```

lookupByKey Dispute During Validation

```
Disputed: recreated and original transaction mismatch VersionedTransaction(...)
↪expected, but VersionedTransaction(...) is recreated.
```

Avoid Race Conditions and Stale References

The first thing to avoid is write-write or write-read contention on contracts. In other words, one requester submitting a transaction with a consuming exercise on a contract while another requester submits another exercise or fetch on the same contract. This type of contention cannot be eliminated entirely, for there will always be some latency between a client submitting a command to a participant, and other clients learning of the committed transaction.

Here are a few scenarios and measures you can take to reduce this type of collision:

1. Shard data. Imagine you want to store a user directory on the Ledger. At the core, this is of type `[(Text, Party)]`, where `Text` is a display name and *Party* the associated Party. If you store this entire list on a single contract, any two users wanting to update their display name at the same time will cause a collision. If you instead keep each `(Text, Party)` on a separate contract, these write operations become independent from each other.
   The Analogy to keep in mind when structuring your data is that a template defines a table, and a contract is a row in that table. Keeping large pieces of data on a contract is like storing big blobs in a database row. If these blobs can change through different actions, you get write conflicts.
2. Use nonconsuming choices if you can. Nonconsuming exercises have the same contention properties as fetches: they don't collide with each other.
   Contract keys can seem like a way out, but they are not. Contract keys are resolved to Contract IDs during the interpretation phase on the participant node. So it reduces latencies slightly by moving resolution from the client layer to the participant layer, but it doesn't remove the issue. Going back to the auction example above, if Alice sent a command `exerciseByKey @Auction auctionKey Bid with amount = 100`, this would be resolved to an `exercise cid Bid with amount = 100` during interpretation, where `cid` is the participant's best guess what ContractId the key refers to.
3. Avoid workflows that encourage multiple parties to simultaneously try to exercise a consuming choice on the same contract. For example, imagine an `Auction` contract containing a field `highestBid : (Party, Decimal)`. If Alice tries to bid $100 at the same time that Bob tries to bid $90, it doesn't matter that Alice's bid is higher. The second transaction to be sequenced will be rejected as it has a write collision with the first. It's better to record the bids in separate `Bid` contracts, which can be written to independently. Again, think about how you would structure this data in a relational database to avoid data loss due to race conditions.
4. Think carefully about storing ContractIds. Imagine you had created a sharded user directory according to 1. Each user has a `User` contract that store their display name and party. Now you write a chat application where each `Message` contract refers to the sender by `ContractId User`. If the user changes their display name, that reference goes stale. You either have to modify all messages that user ever sent, or become unable to use the sender contract in Daml. If you need to be able to make this link inside Daml, Contract Keys help here. If the only place you need to link `Party` to `User` is the UI, it might be best to not store contract references in Daml at all.

Collisions Due to Ignorance

The *Daml Ledger Model* specifies authorization rules, and privacy rules. Ie it specifies what makes a transaction conformant, and who gets to see which parts of a committed transaction. It does *not* specify how a command is translated to a transaction. This may seem strange at first since the commands - create, exercise, exerciseByKey, createAndExercise - correspond so closely to actions in the ledger model. But the subtlety comes in on the read side. What happens when the participant, during interpretation, encounters a `fetch`, `fetchByKey`, or `lookupByKey`?

To illustrate the problem, let's assume there is a template `T` with a contract key, and Alice has witnessed two `Create` nodes of a contract of type `T` with key `k`, but no corresponding archive nodes. Alice may not be able to order these two nodes causally in the sense of one create came before the other . See *Causality and Local Daml Ledgers* for an in-depth treatment of causality on Daml Ledgers.

So what should happen now if Alice's participant encounters a `fetchByKey @T k` or `lookupByKey @T k` during interpretation? What if it encounters a `fetch` node? These decisions are part of the operational semantics, and the decision of what should happen is based on the consideration that the chance of a participant submitting an invalid transaction should be minimized.

If a `fetch` or `exercise` is encountered, the participant resolves the contract as long as it has not witnessed an archive node for that contract - ie as long as it can't guarantee that the contract is no longer active. The rationale behind this is that `fetch` and `exercise` use ContractIds, which need to come from somewhere: Command arguments, Contract arguments, or key lookups. In all three cases, someone believes the ContractId to be active still so it's worth trying.

If a `fetchByKey` or `lookupByKey` node is encountered, the contract is only resolved if the requester is a stakeholder on an active contract with the given key. If that's not the case, there is no reason to believe that the key still resolves to some contract that was witnessed earlier. Thus, when using contract keys, make sure you make the likely requesters of transactions observers on your contracts. If you don't, `fetchByKey` will always fail, and `lookupByKey` will always return `None`.

Let's illustrate how collisions and operational semantics and interleave:

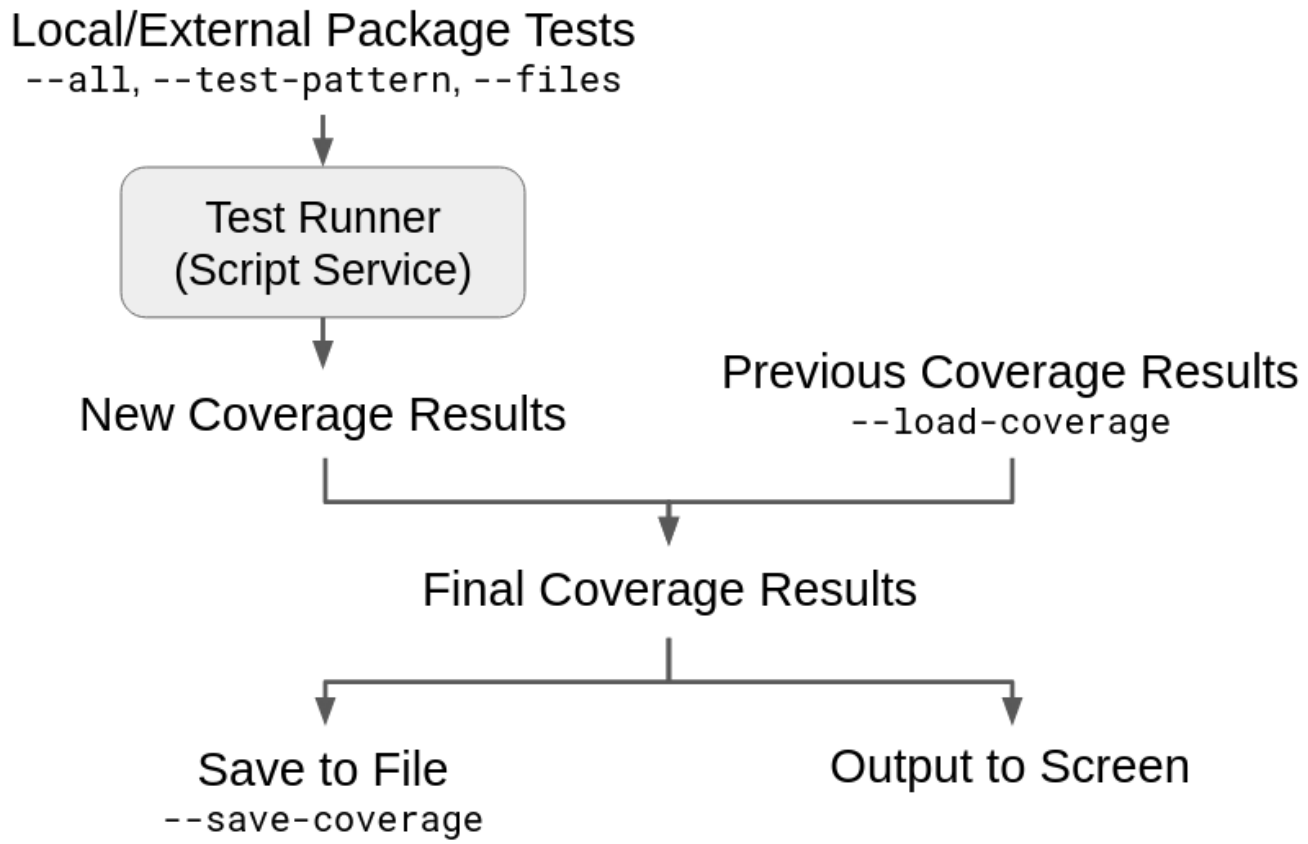
1. Bob creates `T` with key `k`. Alice is not a stakeholder.
2. Alice submits a command resulting in well-authorized `lookupByKey @T k` during interpretation. Even if Alice witnessed 1, this will resolve to a `None` as Alice is not a stakeholder. This transaction is invalid at the time of interpretation, but Alice doesn't know that.
3. Bob submits an `exerciseByKey @T k Archive`.
4. Depending on which of the transactions from 2 and 3 gets sequenced first, either just 3, or both 2 and 3 get committed. If 3 is committed before 2, 2 becomes valid while in transit.

As you can see, the behavior of `fetch`, `fetchByKey` and `lookupByKey` at interpretation time depend on what information is available to the requester at that time. That's something to keep in mind when writing Daml contracts, and something to think about when encountering frequent Disputed errors.

1.11.15.4 Checking Coverage

When `daml test` runs a set of tests, it analyzes the ledger record from those tests to report template and choice coverage. It calculates what percentage of templates defined in the package were created and what percentage of choices defined in the package were exercised.

You can also save the resulting coverage results for the test set to a file and then read them back into a future report. In an invocation of `daml test`, you can both read results in and run tests simultaneously in order to generate a final report which aggregates them. More details on the workflows that this enables are detailed in *Serializing Results Workflows*.



Flags Controlling Test Set

You can control the set of tests run by `daml test` using `--test-pattern PATTERN`, `--files FILE`, and `--all`.

> Passing `--test-pattern <PATTERN>` runs only the local tests which match `PATTERN`.
> Passing `--files <FILE>` runs only the tests found in `FILE`.
> Enabling `--all` runs tests in dependency modules as well. Note: all external tests are run, regardless of the setting in `test-pattern`; `test-pattern` only restricts local tests.

Flags Controlling Serialization

You can save the final coverage results of a `daml test` invocation using `--save-coverage FILE`. This writes the list of templates and choices in scope, along with the list of templates created and choices exercised.

You can read in previous coverage results using `--load-coverage FILE`. This flag can be set multiple times, in which case the results from each file will be read and aggregated into the final result.

There may be occasions where you only need to aggregate coverage results from files, without running any tests. To do that, use the `--load-coverage-only` flag, which ensures that no tests are run.

Flags Controlling Report

Enabling `--show-coverage` tells the final printed report to include the names of any templates, choices, and interfaces which are not covered. By default, the report only reports the percentage of coverage.

You can remove choices from the rendered coverage report with `--coverage-ignore-choice PATTERN`. This flag's behavior is further documented in *Excluding Choices from the Coverage Report*.

Define templates, choices, and interfaces

To demonstrate how the coverage report works, we start by defining three dummy templates, `T1`, `T2`, and `T3`. Each template has two dummy choices:

```
-- Create three dummy tokens with two dummy choices each
template T1 with owner : Party where
  signatory owner

  nonconsuming choice C_T1_1 : ()
    controller owner
    do pure ()

  nonconsuming choice C_T1_2 : ()
    controller owner
    do pure ()

template T2 with owner : Party where
  signatory owner

  nonconsuming choice C_T2_1 : ()
    controller owner
    do pure ()

  nonconsuming choice C_T2_2 : ()
    controller owner
    do pure ()

template T3 with owner : Party where
  signatory owner
```

(continues on next page)

(continued from previous page)

```
  nonconsuming choice C_T3_1 : ()
    controller owner
    do pure ()

  nonconsuming choice C_T3_2 : ()
    controller owner
    do pure ()
```

We also define an interface `I` with instances for `T1` and `T2`:

```
-- Create dummy interface with two dummy choices, implement over T1 and T2, and
↪an unused empty view
data IView = IView {}
interface I where
  viewtype IView
  getController : Party

  nonconsuming choice C_I_1 : ()
    controller (getController this)
    do pure ()

  nonconsuming choice C_I_2 : ()
    controller (getController this)
    do pure ()

  interface instance I for T1 where
    view = IView
    getController = owner

  interface instance I for T2 where
    view = IView
    getController = owner
```

Start testing

By writing a test which selectively creates and exercises only some of these templates and choices, we will see how the coverage report shows us templates and choices we haven't created and exercised respectively.

To start, the test allocates a single party, `alice`, which we will use for the whole test:

```
main = do
  -- Allocate a party
  alice <- allocateParty "Alice"
```

Template creation coverage

The coverage report mentions which templates were defined but never created. For example, the following test creates contracts out of only `T1` and `T2`, never creating instances of template `T3`:

```
  -- Create contracts out of templates T1 and T2
  t1 <- submit alice (createCmd T1 with owner = alice)
  t2 <- submit alice (createCmd T2 with owner = alice)
```

Running `daml test --show-coverage` reports how many templates were defined (3), how many were created (2, 66.7%), and the names of those that weren't created (`T3`):

```
> daml test --show-coverage
...
Modules internal to this package:
- Internal templates
  3 defined
  2 ( 66.7%) created
  internal templates never created: 1
    Token_Coverage_Part1:T3
...
```

Template choice exercise coverage

The coverage report also tracks which choices were exercised. For example, the following test exercises the first and second choices of `T1` and the second choice of `T2`. It also archives `T1`, but not `T2`.

```
  -- Exercise all choices & archive t1
  submit alice (exerciseCmd t1 C_T1_1)
  submit alice (exerciseCmd t1 C_T1_2)
  submit alice (archiveCmd t1)

  -- Exercise only first choice on t2, don't archive
  submit alice (exerciseCmd t2 C_T2_1)
```

`daml test --show-coverage` reports that the test exercised 4 out of 9 choices, and lists the choices that weren't exercised, including the second choice of `T2` and all the choices on `T3`.

Note that `Token_Coverage_Part1:T2:Archive` is included in the list of unexercised choices - because `t2` was not archived, its `Archive` choice was not run.

```
> daml test --show-coverage
...
- Internal template choices
9 defined
4 ( 44.4%) exercised
internal template choices never exercised: 5
  Token_Coverage_Part1:T2:Archive
  Token_Coverage_Part1:T2:C_T2_2
  Token_Coverage_Part1:T3:Archive
  Token_Coverage_Part1:T3:C_T3_1
  Token_Coverage_Part1:T3:C_T3_2
...
```

Interface choice exercise coverage

The coverage report also tracks interfaces, with two differences: * Because interfaces are not created directly but rather cast from templates which implement them, the coverage report cannot not track their creation nor their archival. * Because interfaces can be cast from many possible implementing templates, the report tracks interface choices by what interface they are exercised on and which template they were cast from. In the report, these interface choices are formatted as `<module>:<template>:<choice_name>` - the `<choice_name>` tells us the interface, the `<template>` tells us the template type an interface contract was cast from.

The following test creates `t1` and `t2` as before, but casts them immediately to `I` to get two contracts of `I`: `t1_i` via `T1`, and `t2_i` via `T2`. It exercises both choices on the `t1_i`, but only the first choice on `t2_i`.

```
  -- Exercise all choices on t1_i
  submit alice (exerciseCmd t1_i C_I_1)
  submit alice (exerciseCmd t1_i C_I_2)

  -- Exercise only first choice on t2_i
  submit alice (exerciseCmd t2_i C_I_1)
```

In the coverage report, there are four detected choices, as expected: two choices for the implementation of `I` for `T1`, and two choices for the implementation of `I` for `T2`. Three were exercised, so the only choice that wasn't exercised was `C_I_1` for `T2`, which is reported as `Token_Coverage_Part1:T2:C_I_1`.

```
> daml test --show-coverage
...
- Internal interface choices
  4 defined
  3 ( 75.0%) exercised
  internal interface choices never exercised: 1
    Token_Coverage_Part1:T2:C_I_2
...
```

1.11.15.5 Checking Coverage of External Dependencies

The coverage report also describes coverage for external templates, interfaces, and choices. In the `intro12-part1` directory, run `daml build --output intro12-part1.dar`, and copy the resulting `./intro12-part1.dar` file into the `intro12-part2` directory, where the remainder of our commands will be run.

The `daml.yaml` configuration file in part2 specifies `intro12-part1.dar` as a dependency, letting us import its module.

Definitions

We begin by defining importing the external dependency `Token_Coverage_Part1` as `External` to bring all of its external templates and interfaces into scope.

```
import qualified Token_Coverage_Part1 as External
```

We also define a dummy template `T` with no choices, but an implementation of external interface `External.I`.

```
template T with owner: Party where
  signatory owner

  interface instance External.I for T where
    view = External.IView
    getController = owner
```

Finally, we define an interface `I` with one dummy choice, and implementations for our local template `T` and the external template `External.T1`.

```
data IView = IView {}
interface I where
  viewtype IView
  getController : Party

  nonconsuming choice I_C : ()
    controller (getController this)
    do pure ()

  interface instance I for T where
    view = IView
    getController = owner

  interface instance I for External.T1 where
    view = IView
    getController = owner
```

Local Definitions

Running `daml test -p '^$'` to create a coverage report without running any tests: Because no tests were run, coverage will be 0% in all cases. However, the report will still tally all discovered templates, interfaces, and choices, both external and internal.

```
Modules internal to this package:
- Internal templates
  1 defined
  0 (  0.0%) created
- Internal template choices
  1 defined
  0 (  0.0%) exercised
- Internal interface implementations
  3 defined
    2 internal interfaces
    1 external interfaces
```

(continues on next page)

(continued from previous page)

```
- Internal interface choices
  4 defined
  0 (  0.0%) exercised

Modules external to this package:
- External templates
  3 defined
  0 (  0.0%) created in any tests
  0 (  0.0%) created in internal tests
  0 (  0.0%) created in external tests
- External template choices
  9 defined
  0 (  0.0%) exercised in any tests
  0 (  0.0%) exercised in internal tests
  0 (  0.0%) exercised in external tests
- External interface implementations
  2 defined
- External interface choices
  4 defined
  0 (  0.0%) exercised in any tests
  0 (  0.0%) exercised in internal tests
  0 (  0.0%) exercised in external tests
```

We defined 1 template with 1 default choice (`Archive`), which get reported along with their coverage in the first two sections:

```
- Internal templates
  1 defined
  0 (  0.0%) created
- Internal template choices
  1 defined
  0 (  0.0%) exercised
```

We also have 3 interface implementations that we have defined locally, `External.I for T`, `I for T`, and `I for External.T1`. Note that while the interface implementations are local, the interfaces that they are defined over can be non-local - in this case we have 2 for the local interface `I`, and 1 for the external interface `External.I`. The total number of locally defined implementations, and the breakdown into local interfaces and external interfaces, is presented in the Internal interface implementations section.

```
- Internal interface implementations
  3 defined
    2 internal interfaces
    1 external interfaces
```

These local interface implementations provide 4 choices, two from `External.I for T`, one from `I for T`, and one from `I for External.T1`, reported in the next section along with coverage.

```
- Internal interface choices
  4 defined
  0 (  0.0%) exercised
```

External Definitions

By importing `Token_Coverage_Part1` as `External`, we have brought 3 templates, 9 template choices, 2 interface instances, and 4 interface choices into scope from there, which are listed in the external modules section.

```
...
Modules external to this package:
- External templates
  3 defined
  ...
- External template choices
  9 defined
  ...
- External interface implementations
  2 defined
- External interface choices
  4 defined
  ...
```

External, Internal, and "Any" Coverage

Unlike internal types, externally defined types can be covered by both internal and external tests. As a result, the report for external types distinguishes between coverage provided by internal tests, external tests, and any tests (both internal and external tests).

Here we cover how to run internal and external tests simultaneously to get an aggregate report, and how to interpret this report.

The `--all` flag runs tests in external modules as well. Run `daml test --all --test-pattern notests` in the `intro12-part2` directory - this instructs `daml test` to run all tests from external modules, and to run local tests matching `notests`. We have no local tests named `notests`, so this will only run the `main` test from `part1`. Because the `main` test from `part1` does not use any of the types defined in `part2`, the internal section of the resulting coverage report shows 0% everywhere. However, the `main` test does exercise many types in `part1` which are external to `part2` - as a result, the report's external section is similar to the internal section in the report for `part1`:

```
...
Modules external to this package:
- External templates
  3 defined
  2 ( 66.7%) created in any tests
  0 (  0.0%) created in internal tests
  2 ( 66.7%) created in external tests
- External template choices
  9 defined
  4 ( 44.4%) exercised in any tests
  0 (  0.0%) exercised in internal tests
  4 ( 44.4%) exercised in external tests
- External interface implementations
  2 defined
- External interface choices
  4 defined
  3 ( 75.0%) exercised in any tests
```

(continues on next page)

(continued from previous page)

```
  0 (  0.0%) exercised in internal tests
  3 ( 75.0%) exercised in external tests
```

Note that unlike the internal section of the report in Part 1, the external section of the report in Part 2 has coverage for internal tests, external tests, and any tests. In this report, we only ran an external test, `External:main`, so 0 is reported for all internal tests.

Let's write a local test which will create `External:T3`, a template which the `External:main` test does not create.

```
testT3 : Script ()
testT3 = do
  alice <- allocateParty "Alice"
  external_t3 <- submit alice (createCmd External.T3 with owner = alice)
  pure ()
```

If we run this test on its own using `daml test --test-pattern testT3`, our external coverage report will show that 1 out of 3 of the external templates in scope were created, 1 by internal tests and 0 by external tests.

```
...
modules external to this package:
- external templates
  3 defined
  1 ( 33.3%) created in any tests
  1 ( 33.3%) created in internal tests
  0 (  0.0%) created in external tests
...
```

We can run this test alongside the `External:main` test using `daml test --all --test-pattern testT3`, to get an aggregate coverage report. The report now shows that 2 out of 3 of the external templates in scope were created in `External:main`, and 1 out of 3 by internal test `testT3`. Because `External:main` creates `External:T1` and `External:T2`, and `testT3` creates `External:T3`, all types are created across our tests, and the overall coverage across any tests is 3 out of 3 (100%).

```
...
Modules external to this package:
- External templates
  3 defined
  3 (100.0%) created in any tests
  1 ( 33.3%) created in internal tests
  2 ( 66.7%) created in external tests
...
```

If we define a different local test, `testT1AndT2`, which creates `T1` and `T2`, running it alongside `External:Main`, our report shows 2 out of 3 for internal tests, 2 out of 3 for external tests, but 2 out of 3 for any tests! Because the templates created by each test overlap, `T3` is never created and never covered, so despite an abundance of testing for external templates, coverage is still less than 100%.

```
testT1AndT2 : Script ()
testT1AndT2 = do
  alice <- allocateParty "Alice"
```

(continues on next page)

(continued from previous page)

```
  external_t1 <- submit alice (createCmd External.T1 with owner = alice)
  external_t2 <- submit alice (createCmd External.T2 with owner = alice)
  pure ()
```

```
...
Modules external to this package:
- External templates
  3 defined
  2 ( 66.7%) created in any tests
  2 ( 66.7%) created in internal tests
  2 ( 66.7%) created in external tests
...
```

External template choices and interface instance choices are also reported with `any`, internal, and external coverage - we will not cover them here.

Serializing Results Workflows

The `--save-coverage` and `--load-coverage` flags enable you to write coverage results to a file and read them back out again. Multiple coverage results from different files can be aggregated, along with new coverage results from tests, and then written back to a new file.

This enables three new kinds of coverage testing which should be especially useful to those with large test suites.

Single Test Iteration

When iterating on a single test, you can see the overall coverage of the system by loading in coverage results from all unchanged tests and running the single test, producing an aggregate result.

```
> # Run tests 1 through 8, long running
> daml test --pattern Test[12345678] --save-coverage unchanged-test-results
...
> # Run test 9, aggregate with results from tests 1 through 8
> daml test --pattern Test9 --load-coverage unchanged-test-results
...
> # ... make some changes to test 9 ...
> # Only need to run test 9 to compare coverage report
> daml test --pattern Test9 --load-coverage unchanged-test-results
```

Multiple Test Aggregation

When running a large test suite, you can split the suite across multiple machines and aggregate the results.

```
> # On machine 1:
> daml test --pattern Machine1Test --save-coverage machine1-results
...
> # On machine 2:
```

(continues on next page)

(continued from previous page)

```
> daml test --pattern Machine2Test --save-coverage machine2-results
...
> # On machine 3:
> daml test --pattern Machine3Test --save-coverage machine3-results
...
> # Aggregate results into a single report once all three are done
> daml test --load-coverage-only \
    --load-coverage machine1-results \
    --load-coverage machine2-results \
    --load-coverage machine3-results
```

Test Failure Recovery

If a test failure causes one `daml test` to fail, other coverage results from other tests can be used, and only the failing test needs to be rerun.

```
> # First test run
> daml test --pattern Test1 --save-coverage test1-results
...
> # Second test run:
> daml test --pattern Test2 --save-coverage test2-results
...
> # Third test run (failing):
> daml test --pattern Test3 --save-coverage test3-results
...
FAILED
...
> # ... fix third test ...
> # Third test run (succeeds):
> daml test --pattern Test3 --save-coverage test3-results
...
> # Aggregate results into a single report once all three are done
> daml test --load-coverage-only \
    --load-coverage test1-results \
    --load-coverage test2-results \
    --load-coverage test3-results
```

1.11.15.6 Excluding Choices from the Coverage Report

To exclude choices from the printed coverage report, use `--coverage-ignore-choice PATTERN`. Any choice whose fully qualified name matches the regular expression in `PATTERN` is removed from the coverage report. The choice will not be included in counts of defined choices or in counts of exercised choices. The choice is treated as if it does not exist.

The fully qualified name of a choice depends on whether the choice is defined in the local package or in an external package. Choices defined in the local package are fully qualified as `<module>:<template>:<choice name>`. Choices defined in external packages are fully qualified as `<package id>:<module>:<template>:<choice name>`. By defining your pattern to match different sections in the fully qualified names of your choices, you can exclude choices based on package id, module, template, or name.

Example: Excluding Archive Choices

To exclude the `Archive` choice from coverage, match for the string Archive in the name portion of the fully qualified name. Do this by specifying `--coverage-ignore-choice ':Archive$'`.

If applied to the coverage report in *Template choice exercise coverage*, your coverage report changes from the following:

```
> daml test --show-coverage
...
- Internal template choices
9 defined
4 ( 44.4%) exercised
internal template choices never exercised: 5
  Token_Coverage_Part1:T2:Archive
  Token_Coverage_Part1:T2:C_T2_2
  Token_Coverage_Part1:T3:Archive
  Token_Coverage_Part1:T3:C_T3_1
  Token_Coverage_Part1:T3:C_T3_2
...
```

to a report that ignores `Archive` choices in all cases:

```
> daml test --show-coverage --coverage-ignore-choice ':Archive$'
...
- Internal template choices
7 defined
4 ( 57.1%) exercised
internal template choices never exercised: 3
  Token_Coverage_Part1:T2:C_T2_2
  Token_Coverage_Part1:T3:C_T3_1
  Token_Coverage_Part1:T3:C_T3_2
...
```

Example: Excluding Choices from a Specific Module

To exclude a specific module (for example `MyModule`) from coverage, match for the module portion of the fully qualified name. Do this by specifying `--coverage-ignore-choice '(^|:)MyModule:[^:]*:[^:]*$'`. This matches for any template and any choice, matches for your module name, and ignores any leading package identifier.

Excluding Choices from Serialized Reports

To ensure that serialized data always reflects full coverage information, the flag does **not** eliminate the choices from serialization using the `--save-coverage` flag. Serialized reports saved to a file always contain all information collected. The `--coverage-ignore-choice` flag only excludes choices from the printed report. For any text report generated from serialized data, you must specify `--coverage-ignore-choice` every time it is generated.

1.11.15.7 Next Up

There's little more to learn about writing Daml at this point that isn't best learned by practice and consulting reference material for both Daml and Haskell. In section 13, *Interfaces* we will cover the use of interfaces, a feature which aids code reuse across your Daml programs.

1.11.16 Next Steps

Now that you have completed this introduction to the Daml smart contract language, where do you go next? It depends on what you would like to do with Daml:

What you have learned so far should be enough to enable to you become a certified Daml modeler. You can test your skills at Daml certifications.
If you want to improve your understanding of proven design patterns, you can learn more at *the Patterns* page.
If you're interested in building off-ledger services that interact and integrate with your on-ledger Daml models, read the *Building Applications section*.
If you're interested in understanding how to install, operate and maintain a production-grade Daml ledger, you can have a look at the Canton user manual.
If you want to build Daml applications in a fully-managed environment that handles the day-to-day operation of your Daml ledger for you, you can start right away on Daml Hub.
If you want want to see more examples of Daml applications to understand what is possible with Daml, we have a a library full of examples for you to study.

1.12 Integrate Daml with Off-Ledger Services

1.12.1 Building Applications

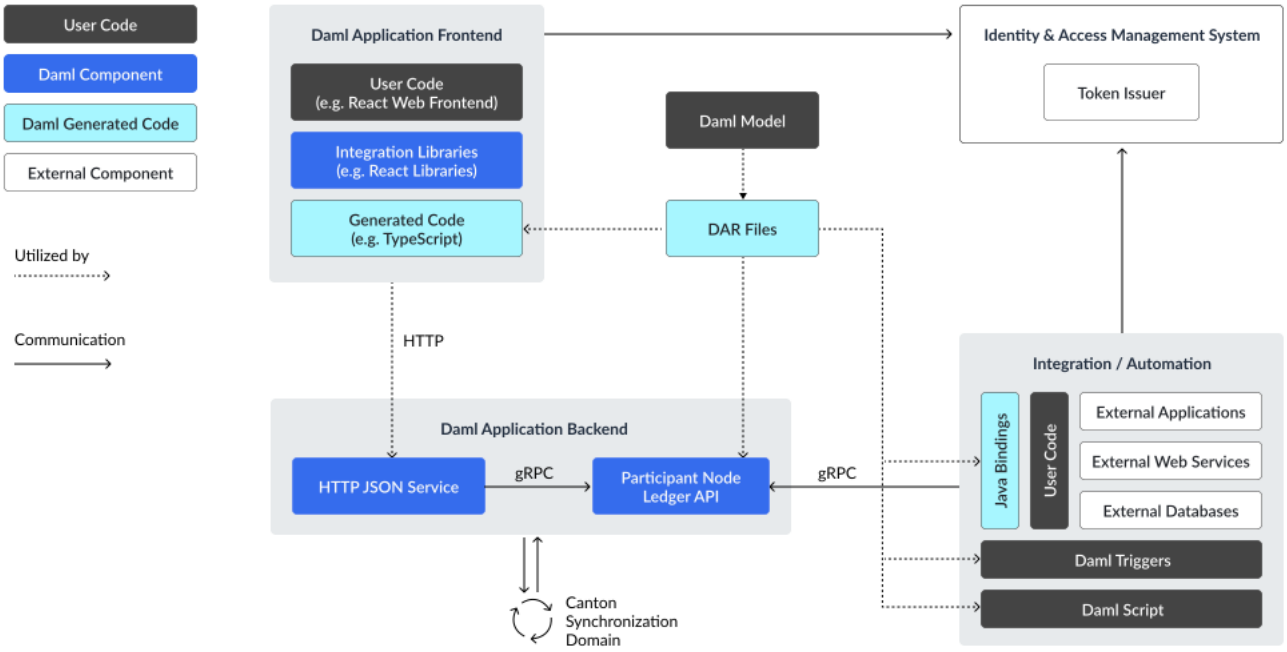
The Building Applications section covers the elements that are used to create, extend, and test your Daml full-stack application (including APIs and JavaScript client libraries) and the architectural best practices for bringing those elements together.

As with the Writing Daml section, you can find the Daml code for the example application and features here or download it using the Daml assistant. For example, to load the sources for section 1 into a folder called `intro1`, run daml new intro1 –template daml-intro-1.

To run the examples, you will first need to install the Daml SDK.

1.12.2 Daml Application Architecture

This section describes our recommended design of a full-stack Daml application.



The above image shows the recommended Daml solution architecture. Here there are four types of building blocks that go into our application: user code, generated code from Daml, Daml components, and external components.

In the recommended architecture, the Daml model determines the DAR files that underpin both the frontend and backend. The frontend includes user code such as a React Web Frontend, Daml React libraries or other integration libraries, and generated code from the DAR files (TypeScript). A client service can access the Daml application backend instead of a GUI frontend with no change to the rest of the architecture.

From the client point of view, the Daml application backend consists of the JSON API and a participant node. The backend uses a Canton synchronization domain (not shown) to distribute changes to the ledger made by the application, as well as changes made by other applications, to all domain-connected participants.

Integrations with a Daml application are done via Java bindings, while automation can be done with Daml Script and/or Daml Triggers. Daml Scripts allows you to write automations that can be triggered by any off-ledger condition, such as the availability of a file in a folder or a message coming from a broker or a user interacting with the system directly. Daml Triggers allow a similar approach but are triggered by on-ledger events, such as the creation of a contract.

Daml application uses JWT tokens for access authorization, checking if the party submitting the request has the necessary rights for it. How an application acquires access tokens depends on the participant node it talks to and is ultimately set up by the participant node operator.

There are many ways that the architecture and technology stack can be changed to fit your needs, which we'll mention in the corresponding sections.

To get started quickly with the recommended application architecture, generate a new project using the `create-daml-app` template:

```
daml new --template=create-daml-app my-project-name
```

`create-daml-app` is a small, but fully functional demo application implementing the recommended architecture, providing you with an excellent starting point for your own application. It

showcases

using Daml React libraries
quick iteration against the *Daml Sandbox*.
authorization
deploying your application in the cloud as a Docker container

1.12.2.1 Backend

The backend for your application can be any Daml ledger implementation running your DAR (*Daml Archive*) file.

We recommend using the *Daml JSON API* as an interface to your frontend. It is served by the HTTP JSON API server connected to the ledger API server. It provides simple HTTP endpoints to interact with the ledger via GET/POST requests. However, if you prefer, you can also use the *gRPC Ledger API* directly.

When you use the `create-daml-app` template application, you can start a Daml Sandbox together with a JSON API server by running the following command in the root of the project.

```
daml start --start-navigator=no
```

Daml Sandbox exposes the same Daml Ledger API a Participant Node would expose without requiring a fully-fledged Daml network to back the application. Once your application matures and becomes ready for production, the `daml deploy` command helps you deploy your frontend and Daml artifacts of your project to a production Daml network.

1.12.2.2 Frontend

We recommended building your frontend with the React framework. However, you can choose virtually any language for your frontend and interact with the ledger via *HTTP JSON* endpoints. In addition, we provide support libraries for *Java* and you can also interact with the *gRPC Ledger API* directly.

We provide two libraries to build your React frontend for a Daml application.

Name	Summary
@daml/react	React hooks to query/create/exercise Daml contracts
@daml/ledger	Daml ledger object to connect and directly submit commands to the ledger

You can install any of these libraries by running `npm install <library>` in the `ui` directory of your project, e.g. `npm install @daml/react`. Please explore the `create-daml-app` example project to see the usage of these libraries.

To make your life easy when interacting with the ledger, the Daml assistant can generate JavaScript libraries with TypeScript typings from the data types declared in the deployed DAR.

```
daml codegen js .daml/dist/<your-project-name.dar> -o ui/daml.js
```

This command will generate a JavaScript library for each DALF in your DAR, containing metadata about types and templates in the DALF and TypeScript typings them. In `create-daml-app`, `ui/package.json` refers to these libraries via the `"create-daml-app": "file:../daml.js/create-daml-app-0.1.0"` entry in the `dependencies` field.

If you choose a different JavaScript based frontend framework, the packages `@daml/ledger`, `@daml/types` and the generated `daml.js` libraries provide you with the necessary code to connect and issue commands against your ledger.

1.12.2.3 Authorization

When you deploy your application to a production ledger, you need to authenticate the identities of your users.

Daml ledgers support a unified interface for authorization of commands. Some Daml ledgers, like for example https://hub.daml.com, offer integrated authentication and authorization, but you can also use an external service provider like https://auth0.com. The Daml react libraries support interfacing with a Daml ledger that validates authorization of incoming requests. Simply initialize your `DamlLedger` object with the token obtained by the respective token issuer. How authorization works and the form of the required tokens is described in the *Authorization* section.

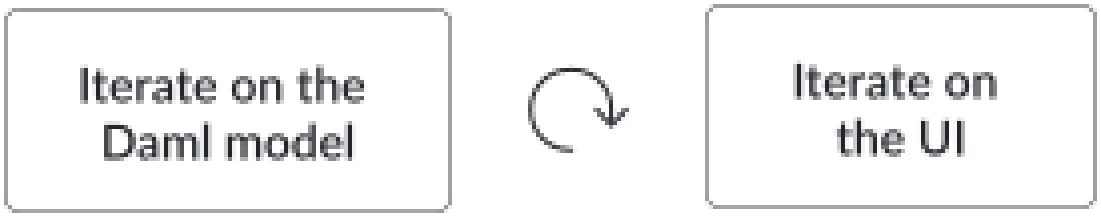
1.12.2.4 Developer Workflow

The SDK enables a local development environment with fast iteration cycles:

1. The integrated VSCode IDE (`daml studio`) runs your Scripts on any change to your Daml models. See *Daml Script*.
2. `daml start` will build all of your Daml code, generate the JavaScript bindings, and start the required backend processes (sandbox and HTTP JSON API). It will also allow you to press `r` (followed by Enter on Windows) to rebuild your code, regenerate the JavaScript bindings and upload the new code to the running ledger.
3. `npm start` will watch your JavaScript source files for change and recompile them immediately when they are saved.

Together, these features can provide you with very tight feedback loops while developing your Daml application, all the way from your Daml contracts up to your web UI. A typical Daml developer workflow is to

1. Make a small change to your Daml data model
2. Optionally test your Daml code with *Daml Script*
3. Edit your React components to be aligned with changes made in Daml code
4. Extend the UI to make use of the newly introduced feature
5. Make further changes either to your Daml and/or React code until you're happy with what you've developed



See *Your First Feature* for a more detailed walkthrough of these steps.

Command Deduplication

The interaction of a Daml application with the ledger is inherently asynchronous: applications send commands to the ledger, and some time later they see the effect of that command on the ledger.

Several things can fail during this time window: the application can crash, the participant node can crash, messages can be lost on the network, or the ledger may be just slow to respond due to a high load.

If you want to make sure that a command is not executed twice, your application needs to robustly handle all failure scenarios. Daml ledgers provide a mechanism for *command deduplication* to help deal with this problem.

For each command the application provides a command ID and an optional parameter that specifies the deduplication period. If the latter parameter is not specified in the command submission itself, the ledger will use the configured maximum deduplication duration. The ledger will then guarantee that commands with the same *change ID* will generate a rejection within the effective deduplication period.

For details on how to use command deduplication, see the *Command Deduplication Guide*.

Deal With Failures

Crash Recovery

In order to restart your application from a previously known ledger state, your application must keep track of the last ledger offset received from the *transaction service* or the *command completion service*.

By persisting this offset alongside the relevant state as part of a single, atomic operation, your application can resume from where it left off.

Fail Over Between Ledger API Endpoints

Some Daml Ledgers support exposing multiple eventually consistent Ledger API endpoints where command deduplication works across these Ledger API endpoints. For example, these endpoints might be hosted by separate Ledger API servers that replicate the same data and host the same parties. Contact your ledger operator to find out whether this applies to your ledger.

Below we describe how you can build your application such that it can switch between such eventually consistent Ledger API endpoints to tolerate server failures. You can do this using the following two steps.

First, your application must keep track of the ledger offset as described in the *paragraph about crash recovery*. When switching to a new Ledger API endpoint, it must resume consumption of the transaction (tree) and/or the command completion streams starting from this last received offset.

Second, your application must retry on `OUT_OF_RANGE` errors (see gRPC status codes) received from a stream subscription – using an appropriate backoff strategy to avoid overloading the server. Such errors can be raised because of eventual consistency. The Ledger API endpoint that the application is newly subscribing to might be behind the endpoint that it subscribed to before the switch, and needs time to catch up. Thanks to eventual consistency this is guaranteed to happen at some point in the future.

Once the application successfully subscribes to its required streams on the new endpoint, it will resume normal operation.

Deal With Time

The Daml language contains a function *getTime* which returns a rough estimate of current time called *Ledger Time*. The notion of time comes with a lot of problems in a distributed setting: different participants might run different clocks, there may be latencies due to calculation and network, clocks may drift against each other over time, etc.

In order to provide a useful notion of time in Daml without incurring severe performance or liveness penalties, Daml has two notions of time: *Ledger Time* and *Record Time*:

> As part of command interpretation, each transaction is automatically assigned a *Ledger Time* by the participant server.
> All calls to `getTime` within a transaction return the *Ledger Time* assigned to that transaction.
> *Ledger Time* is chosen (and validated) to respect Causal Monotonicity: The Create action on a contract *c* always precedes all other actions on *c* in Ledger Time.
> As part of the commit/synchronization protocol of the underlying infrastructure, every transaction is assigned a *Record Time*, which can be thought of as the infrastructures system time . It's the best available notion of real time , but the only guarantees on it are the guarantees the underlying infrastructure can give. It is also not known at interpretation time.
> *Ledger Time* is kept close to real time by bounding it against *Record Time*. Transactions where *Ledger* and *Record Time* are too far apart are rejected.

Some commands might take a long time to process, and by the time the resulting transaction is about to be committed to the ledger, it might violate the condition that *Ledger Time* should be reasonably close to *Record Time* (even when considering the ledger's tolerance interval). To avoid such problems, applications can set the optional parameters *min_ledger_time_abs* or *min_ledger_time_rel* that specify (in absolute or relative terms) the minimal *Ledger Time* for the transaction. The ledger will then process the command, but wait with committing the resulting transaction until *Ledger Time* fits within the ledger's tolerance interval.

How is this used in practice?

> Be aware that `getTime` is only reasonably close to real time, and not completely monotonic. Avoid Daml workflows that rely on very accurate time measurements or high frequency time changes.
> Set `min_ledger_time_abs` or `min_ledger_time_rel` if the duration of command interpretation and transmission is likely to take a long time relative to the tolerance interval set by the ledger.
> In some corner cases, the participant node may be unable to determine a suitable Ledger Time by itself. If you get an error that no Ledger Time could be found, check whether you have contention on any contract referenced by your command or whether the referenced contracts are sensitive to small changes of `getTime`.

For more details, see *Background concepts - time*.

1.12.3 Parties and Users On a Daml Ledger

Identifying parties and users is an important part of building a workable Daml application. Recall these definitions from the *Getting Started Guide*:

> **Parties** are unique across the entire Daml network. These must be allocated before you can use them to log in, and allocation results in a random-looking (but not actually random) string that identifies the party and is used in your Daml code. Parties are a builtin concept.
> On each participant node you can create **users** with human-readable user ids. Each user can be associated with one or more parties allocated on that participant node, and refers to that party only on that node. Users are a purely local concept, meaning you can never address a user on another node by user id, and you never work with users in your Daml code; party ids are always used for these purposes. Users are also a builtin concept.

This represents a change from earlier versions of Daml, and the implications of these changes are discussed in more depth here.

1.12.3.1 Parties in SDK 2.0 and Subsequent

In Daml 2.0 and later versions, when you allocate a party with a given hint Alice either in the sandbox or on a production ledger you will get back a party id like `Alice::1220f2fe29866fd6a0009ecc8a64ccdc09f1958bd0f801166baaee469d1251b2eb72`. The prefix before the double colon corresponds to the hint specified on party allocation. If the hint is not specified, it defaults to `party-${randomUUID}`. The suffix is the fingerprint of the public key that can authorize topology transactions for this party. Keys are generated randomly, so the suffix will look different locally and every time you restart Sandbox, you will get a different party id. This has a few new implications:

> You can no longer allocate a party with a fixed party id. While you have some control over the prefix, we do not recommend that you rely on that to identify parties.
> Party ids are no longer easily understandable by humans. You may want to display something else in your user interfaces.
> Discovering the party ID of other users might get tricky. For example, to follow the user Bob, you cannot assume that their party ID is Bob .

1.12.3.2 Party ID Hints and Display Names

Party id hints and display names which existed in SDK 1.18.0 are still available in SDK 2.0.0. We recommend against relying on display names for new applications, but if you are migrating your existing application, they function exactly as before.

Party id hints still serve a purpose. While we recommend against parsing party ids and extracting the hint, for debugging and during development it can be helpful to see the party id hint at the beginning. Bear in mind that different parties can be allocated to different participants with the same party id hint. The full party ids will be different due to the suffix, but the party id hint would be the same.

The second remaining use for party id hints is to avoid duplicate party allocation. Consider sending a party allocation request that fails due to a network error. The client has no way of knowing whether the party has been allocated. Because a party allocation will be rejected if a party with the given hint already exists, the client can safely send the same request with the same hint, which will either allocate a party if the previous request failed or fail itself. (Note that while this works for Canton,

including Sandbox as well as the VMWare blockchain, it is not part of the ledger API specifications, so other ledgers might behave differently.)

1.12.3.3 Authorization and User Management

Daml 2.0 also introduced *user management*. User management allows you to create users on a participant that are associated with a primary party and a dynamic set of actAs and readAs claims. Crucially, the user id can be fully controlled when creating a user – unlike party ids – and are unique on a single participant. You can also use the user id in *authorization tokens* instead of party tokens that have specific parties in actAs and readAs fields. This means your IAM, which can sometimes be limited in configurability, only has to work with fixed user ids.

However, users are purely local to a given participant. You cannot refer to users or parties associated with a given user on another participant via their user id. You also need admin claims to interact with the user management endpoint for users other than your own. This means that while you can have a user id in place of the primary party of your own user, you cannot generally replace party ids with user ids.

1.12.3.4 Working with Parties

So how do you handle these unwieldy party ids? The primary rule is to treat them as *opaque identifiers*. In particular, don't parse them, don't make assumptions about their format, and don't try to turn arbitrary strings into party ids. The only way to get a new party id is as the result of a party allocation. Applications should never hardcode specific parties. Instead either accept them as inputs or read them from contract or choice arguments.

To illustrate this, we'll go over the tools in the SDK and how this affects them:

Daml Script

In Daml script, allocateParty returns the party id that has been allocated. This party can then be used later, for example, in command submissions. When your script should refer to parties that have been allocated outside of the current script, accept those parties as arguments and pass them in via –input-file. Similarly, if your script allocates parties and you want to refer to them outside of the script, either in a later script or somewhere else, you can store them via –output-file. You can also query the party management and user management endpoints and get access to parties that way. Keep in mind though, this requires admin rights on a participant and there are no uniqueness guarantees for display names. That usually makes querying party and user management endpoints usually only an option for development, and we recommend passing parties as arguments where possible instead.

Daml Triggers

To start a trigger via the trigger service, you still have to supply the party ids for the actAs and readAs claims for your trigger. This could, e.g., come from a party allocation in a Daml script that you wrote to a file via Daml Script's –output-file. Within your trigger, you get access to those parties via getActAs and getReadAs. To refer to other parties, for example when creating a contract, reference them from an existing contract. If there is no contract, consider creating a special configuration template that lists the parties your trigger should interact with outside of your trigger, and query for that template in your trigger to get access to the parties.

Navigator

Navigator presents you with a list of user ids on the participant as login options. Once logged in, you will interact with the ledger as the primary party of that user. Any field that expects a party provides autocompletion, so if you know the prefix (by having chosen the hint), you don't have to remember the suffix. In addition, party ids have been shortened in the Navigator UI so that not all of the id is shown. Clicking on a party identifier will copy the full identifier to the system clipboard, making it easier to use elsewhere.

Java Bindings

When writing an application using the Java bindings, we recommend that you pass parties as arguments. They can either be CLI arguments or JVM properties as used in the :doc: *quickstart-java example <bindings-java/quickstart.html>*.

Create-daml-app and UIs

Create-daml-app and UIs in general are a bit more complex. First, they often need to interact with an IAM during the login. Second, it is often important to have human-readable names in a UI — to go back to an earlier example, a user wants to follow Bob without typing a very long party id.

Logging in is going to depend on your specific IAM, but there are a few common patterns. In create-daml-app, you log in by typing your user id directly and then interacting with the primary party of that user. In an authorized setup, users might use their email address and a password, and as a result, the IAM will provide them with a token for their user id. The approach to discovering party ids corresponding to human-readable uses can also vary depending on privacy requirements and other constraints. Create-daml-app addresses this by writing alias contracts on the ledger with associate human-readable names with the party id. These alias contracts are shared with everyone via a public party.

1.12.4 JSON API

1.12.4.1 HTTP JSON API Service

The **JSON API** provides a significantly simpler way to interact with a ledger than *the Ledger API* by providing *basic active contract set functionality*:

creating contracts,
exercising choices on contracts,
querying the current active contract set, and
retrieving all known parties.

The goal of this API is to get your distributed ledger application up and running quickly, so we have deliberately excluded complicating concerns including, but not limited to:

inspecting transactions,
asynchronous submit/completion workflows,
temporal queries (e.g. active contracts *as of a certain time*), and

For these and other features, use *the Ledger API* instead. The HTTP JSON API service is a proxy, after a fashion, for that API; *there is literally nothing that HTTP JSON API service can do that your own application cannot do via gRPC*.

If you are using this API from JavaScript or TypeScript, we strongly recommend using the JavaScript bindings and code generator rather than invoking these endpoints directly. This will both simplify access to the endpoints described here and (with TypeScript) help to provide the correct JavaScript value format for each of your contracts, choice arguments, and choice results.

As suggested by those bindings, the primary target application for the HTTP JSON API service is a web application, where user actions translate to one or a few ledger operations. It is not intended for high-throughput, high-performance ledger automation; the Ledger API is better suited to such use cases.

We welcome feedback about the JSON API on our issue tracker, or on our forum.

Run the JSON API

Start a Daml Ledger

You can run the JSON API alongside any ledger exposing the gRPC Ledger API you want. If you don't have an existing ledger, you can start an in-memory sandbox:

```
daml new my-project --template quickstart-java
cd my-project
daml build
daml sandbox --wall-clock-time --dar ./.daml/dist/quickstart-0.0.1.dar
```

Start the HTTP JSON API Service

Basic

The most basic way to start the JSON API is with the command:

```
daml json-api --config json-api-app.conf
```

where a corresponding minimal config file is

```
{
  server {
    address = "localhost"
    port = 7575
  }
  ledger-api {
    address = "localhost"
    port = 6865
  }
}
```

This will start the JSON API on port 7575 and connect it to a ledger running on `localhost:6865`.

Note: Your JSON API service should never be exposed to the internet. When running in production the JSON API should be behind a reverse proxy, such as via NGINX.

The full set of configurable options that can be specified via config file is listed below

```
{
  server {
    //IP address that HTTP JSON API service listens on. Defaults to 127.0.0.1.
    address = "127.0.0.1"
    //HTTP JSON API service port number. A port number of 0 will let the system
↪pick an ephemeral port.
    port = 7575
  }
  ledger-api {
    address = "127.0.0.1"
    port = 6865
    tls {
        enabled = "true"
        // the certificate to be used by the server
        cert-chain-file = "cert-chain.crt"
        // private key of the server
        private-key-file = "pvt-key.pem"
        // trust collection, which means that all client certificates will be
↪verified using the trusted
        // certificates in this store. if omitted, the JVM default trust store is
↪used.
        trust-collection-file = "root-ca.crt"
    }
  }

  query-store {
```

(continues on next page)

(continued from previous page)

```
    base-config {
      user = "postgres"
      password = "password"
      driver = "org.postgresql.Driver"
      url = "jdbc:postgresql://localhost:5432/test?&ssl=true"

      // prefix for table names to avoid collisions, empty by default
      table-prefix = "foo"

      // max pool size for the database connection pool
      pool-size = 12
      //specifies the min idle connections for database connection pool.
      min-idle = 4
      //specifies the idle timeout for the database connection pool.
      idle-timeout = 12s
      //specifies the connection timeout for database connection pool.
      connection-timeout = 90s
    }
    // option setting how the schema should be handled.
    // Valid options are start-only, create-only, create-if-needed-and-start and
↪create-and-start
    start-mode = "start-only"
  }



  // Optional interval to poll for package updates. Examples: 500ms, 5s, 10min,
↪1h, 1d. Defaults to 5 seconds
  package-reload-interval = 5s
  //Optional max inbound message size in bytes. Defaults to 4194304.
  max-inbound-message-size = 4194304
  //Optional max inbound message size in bytes used for uploading and downloading
↪package updates. Defaults to the `max-inbound-message-size` setting.
  package-max-inbound-message-size = 4194304
  //Optional max cache size in entries for storing surrogate template id mappings.
↪ Defaults to None
  max-template-id-cache-entries = 1000
  //health check timeout in seconds
  health-timeout-seconds = 5

  //Optional websocket configuration parameters
  websocket-config {
    //Maximum websocket session duration
    max-duration = 120m
    //Server-side heartbeat interval duration
    heartbeat-period = 5s
    //akka stream throttle-mode one of either `shaping` or `enforcing`
    mode = "shaping"
  }

  metrics {
    //Start a metrics reporter. Must be one of "console", "csv:///PATH",
↪"graphite://HOST[:PORT][/METRIC_PREFIX]", or "prometheus://HOST[:PORT]".
    reporter = "console"
    //Set metric reporting interval , examples : 1s, 30s, 1m, 1h
    reporting-interval = 30s
```

(continues on next page)

(continued from previous page)

```
  }

  // DEV MODE ONLY (not recommended for production)
  // Allow connections without a reverse proxy providing HTTPS.
  allow-insecure-tokens = false
  // Optional static content configuration string. Contains comma-separated key-
↪value pairs, where:
  // prefix -- URL prefix,
  // directory -- local directory that will be mapped to the URL prefix.
  // Example: "prefix=static,directory=./static-content"
  static-content {
    prefix = "static"
    directory = "static-content-dir"
  }
}
```

Note: You can also start JSON API using CLI args (example below) however this is now deprecated

```
daml json-api --ledger-host localhost --ledger-port 6865 --http-port 7575
```

Standalone JAR

The `daml json-api` command is great during development since it is included with the SDK and integrates with `daml start` and other commands. Once you are ready to deploy your application, you can download the standalone JAR from Github releases. It is much smaller than the whole SDK and easier to deploy since it only requires a JVM but no other dependencies and no installation process. The JAR accepts exactly the same command line parameters as `daml json-api`, so to start the standalone JAR, you can use the following command:

```
java -jar http-json-2.0.0.jar --config json-api-app.conf
```

Replace the version number `2.0.0` by the version of the SDK you are using.

With Query Store

In production setups, you should configure the HTTP JSON API service to use a PostgreSQL backend as a *Query Store*. The in-memory backend will call the ledger to fetch the entire active contract set for the templates in your query every time so it is generally not recommended to rely on this in production. Note that the query store is a redundant copy of on-ledger data. It is safe to reinitialize the database at any time.

To enable the PostgreSQL backend you can add the `query-store` config block *as described*.

Access Tokens

Each request to the HTTP JSON API Service *must* come with an access token, regardless of whether the underlying ledger requires it or not. This also includes development setups using an unsecured sandbox. The HTTP JSON API Service *does not* hold on to the access token, which will be only used to fulfill the request it came along with. The same token will be used to issue the request to the Ledger API.

The HTTP JSON API Service does not validate the token but may need to decode it to extract information that can be used to fill in request fields for party-specific request. How this happens depends partially on the token format you are using.

Party-specific Requests

Party-specific requests, i.e., command submissions and queries, are subject to additional restrictions. For command submissions the token must provide a proof that the bearer can act on behalf of at least one party (and possibly read on behalf of any number of parties). For queries the token must provide a proof that the bearer can either act and/or read of at least one party. This happens regardless of the used *access token format*. The following paragraphs provide guidance as to how different token formats are used by the HTTP JSON API in this regard.

Using User Tokens

If the underlying ledger supports *user management* (this includes Canton and the sandbox), you are recommended to use user tokens. For command submissions, the user of the bearer should have `actAs` rights for at least one party and `readAs` rights for any number of parties. Queries require the bearer's user to have at least one `actAs` or `readAs` user right. The application id of the Ledger API request will be the user id.

Using Claim Tokens

These tokens can be used if the underlying ledger does not support *user management*. For command submissions, `actAs` must contain at least one party and `readAs` can contain any number of parties. Queries require at least one party in either `actAs` or `readAs`. The application id is mandatory.

Note: While the JSON API receives the token it doesn't validate it itself. Upon receiving a token it will pass it, and all data contained within the request, on to the Ledger API's AuthService which will then determine if the token is valid and authorized. However, the JSON API does decode the token to extract the ledger id, application id and party so it requires that you use *a valid Daml ledger access token format*.

For a ledger without authorization, e.g., the default configuration of Daml Sandbox, you can use https://jwt.io (or the JWT library of your choice) to generate your token. You can use an arbitrary secret here. The default header is fine. Under Payload, fill in:

```
{
  "https://daml.com/ledger-api": {
```

(continues on next page)

(continued from previous page)

```
    "ledgerId": "sandbox",
    "applicationId": "foobar",
    "actAs": ["Alice"]
  }
}
```

The value of the `ledgerId` field has to match the `ledgerId` of your underlying Daml Ledger. For the Sandbox this corresponds to the participant id which by default is just *sandbox*.

Note: The value of `applicationId` will be used for commands submitted using that token.

The value for `actAs` is specified as a list and you provide it with the party that you want to use, such as in the example above which uses `Alice` for a party. `actAs` may include more than just one party as the JSON API supports multi-party submissions.

The party should reference an already allocated party.

Note: As mentioned above the JSON API does not validate tokens so if your ledger runs without authorization you can use an arbitrary secret.

Then the Encoded box should have your **token**, ready for passing to the service as described in the following sections.

Alternatively, here are two tokens you can use for testing:

`{"https://daml.com/ledger-api": {"ledgerId": "sandbox", "applicationId": "HTTP-JSON-API-Gateway", "actAs": ["Alice"]}}`:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJodHRwczovL2RhbWwuY29tL2xlZGdlci1hcGkiOnsibGVkZ2VySWQiOiJzYW5kYm94IiwiYXBwbGljYXRpb25JZ
↪FIjS4ao9yu1XYnv1ZL3t7ooPNIyQYAHY3pmzej4EMCM
```

`{"https://daml.com/ledger-api": {"ledgerId": "sandbox", "applicationId": "HTTP-JSON-API-Gateway", "actAs": ["Bob"]}}`:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJodHRwczovL2RhbWwuY29tL2xlZGdlci1hcGkiOnsibGVkZ2VySWQiOiJzYW5kYm94IiwiYXBwbGljYXRpb25JZ
↪y6iwpnYt-ObtNo_FyLVxMtNTwpJF8uxzNfPELQUVKVg
```

Auth via HTTP

Set HTTP header `Authorization: Bearer paste-jwt-here`

Example:

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJodHRwczovL2RhbWwuY29tL2xlZGdlci1hcGkiOnsibGVkZ2VySWQiOiJNeUxlZGdlciIsImFwcGxpY2F0aW9uS
↪34zzF_fbWv7p60r5s1kKzwndvGdsJDX-W4Xhm4oVdpk
```

Auth via WebSockets

WebSocket clients support a subprotocols argument (sometimes simply called protocols); this is usually in a list form but occasionally in comma-separated form. Check documentation for your WebSocket library of choice for details.

For HTTP JSON requests, you must pass two subprotocols:

```
daml.ws.auth
jwt.token.paste-jwt-here
```

Example:

```
jwt.token.eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJodHRwczovL2RhbWwuY29tL2xlZGdlci1hcGkiOnsibGVkZ2VySWQiOiJNeUxlZGdlciIsImFwcGxpY2F0aW9uS
↪34zzF_fbWv7p60r5s1kKzwndvGdsJDX-W4Xhm4oVdpk
```

HTTP Status Codes

The **JSON API** reports errors using standard HTTP status codes. It divides HTTP status codes into 3 groups indicating:

1. success (200)
2. failure due to a client-side problem (400, 401, 403, 404, 409, 429)
3. failure due to a server-side problem (500, 503)

The **JSON API** can return one of the following HTTP status codes:

200 - OK
400 - Bad Request (Client Error)
401 - Unauthorized, authentication required
403 - Forbidden, insufficient permissions
404 - Not Found
409 - Conflict, contract ID or key missing or duplicated
500 - Internal Server Error
503 - Service Unavailable, ledger server is not running yet or has been shut down
504 - Gateway Timeout, transaction failed to receive its completion within the predefined timeout

When the Ledger API returns an error code, the JSON API maps it to one of the above codes according to the official gRPC to HTTP code mapping.

If a client's HTTP GET or POST request reaches an API endpoint, the corresponding response will always contain a JSON object with a `status` field, and either an `errors` or `result` field. It may also contain an optional `warnings` and/or an optional `ledgerApiError` :

```
{
    "status": <400 | 401 | 403 | 404 | 409 | 500 | 503 | 504>,
    "errors": <JSON array of strings>, | "result": <JSON object or array>,
    ["warnings": <JSON object> ],
    ["ledgerApiError": <JSON object> ]
}
```

Where:

`status` - a JSON number which matches the HTTP response status code returned in the HTTP header,
`errors` - a JSON array of strings, each string represents one error,
`result` - a JSON object or JSON array, representing one or many results,
`warnings` - an optional field with a JSON object, representing one or many warnings.
`ledgerApiError` - an optional field with a JSON object, representing detail of an error if it was originated from Ledger API.

See the following blog post for more details about error handling best practices: REST API Error Codes 101.

See *The Ledger API error codes* for more details about error codes from Ledger API.

Successful Response, HTTP Status: 200 OK

Content-Type: `application/json`
Content:

```
{
    "status": 200,
    "result": <JSON object>
}
```

Successful Response with a Warning, HTTP Status: 200 OK

Content-Type: `application/json`
Content:

```
{
    "status": 200,
    "result": <JSON object>,
    "warnings": <JSON object>
}
```

Failure, HTTP Status: 400 | 401 | 404 | 500

Content-Type: `application/json`
Content:

```
{
    "status": <400 | 401 | 404 | 500>,
    "errors": <JSON array of strings>,
    ["ledgerApiError": <JSON object> ]
}
```

Examples

Result with JSON Object without Warnings:

```
{"status": 200, "result": {...}}
```

Result with JSON Array and Warnings:

```
{"status": 200, "result": [...], "warnings": {"unknownTemplateIds": [
↪"UnknownModule:UnknownEntity"]}}
```

Bad Request Error:

```
{"status": 400, "errors": ["JSON parser error: Unexpected character 'f' at input
↪index 27 (line 1, position 28)"]}
```

Bad Request Error with Warnings:

```
{"status":400, "errors":["Cannot resolve any template ID from request"], "warnings
↪":{"unknownTemplateIds":["XXX:YYY","AAA:BBB"]}}
```

Authentication Error:

```
{"status": 401, "errors": ["Authentication Required"]}
```

Not Found Error:

```
{"status": 404, "errors": ["HttpMethod(POST), uri: http://localhost:7575/v1/query1
↪"]}
```

Internal Server Error:

```
{"status": 500, "errors": ["Cannot initialize Ledger API"]}
```

Create a New Contract

To create an `Iou` contract from the *Quickstart guide*:

```
template Iou
  with
    issuer : Party
    owner : Party
    currency : Text
    amount : Decimal
    observers : [Party]
```

HTTP Request

URL: `/v1/create`
Method: `POST`
Content-Type: `application/json`
Content:

```
{
  "templateId":
↪"a3b788b4dc18dc060bfb82366ae6dc055b1e361d646d5cfdb1b729607e344336:Iou:IouTransfer
↪",
  "payload": {
    "issuer": "Alice",
    "owner": "Alice",
    "currency": "USD",
    "amount": "999.99",
    "observers": []
  }
}
```

Where:

`templateId` is the contract template identifier, which is formatted as "`<package ID>:<module>:<entity>`". As a convenience for interactive API exploration (such as with curl and similar tools), you can also omit the package ID (i.e. specifying the `templateId` as "`<module>:<entity>`") **if there is only one template with that name across all loaded packages**. Code should always specify the package ID, since it's common to have more versions of a template sharing the same module and entity name but with different package IDs. If the package identifier is not specified and the template cannot be uniquely identified without it, the HTTP JSON API service will report that the specified template cannot be found. **Omitting the package ID is not supported for production use.**
`payload` field contains contract fields as defined in the Daml template and formatted according to *Daml-LF JSON Encoding*.

HTTP Response

Content-Type: `application/json`
Content:

```
{
    "status": 200,
    "result": {
        "observers": [],
        "agreementText": "",
        "payload": {
            "observers": [],
            "issuer": "Alice",
            "amount": "999.99",
            "currency": "USD",
            "owner": "Alice"
        },
        "signatories": [
            "Alice"
```

(continues on next page)

(continued from previous page)

```
        ],
        "contractId": "#124:0",
        "templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:Iou",
        "completionOffset":"0000000000000084"
    }
}
```

Where:

`status` field matches the HTTP response status code returned in the HTTP header,
`result` field contains created contract details. Keep in mind that `templateId` in the **JSON API** response is always fully qualified (always contains package ID).

Create a Contract with a Command ID

When creating a new contract or exercising a choice you may specify an optional `meta` field. This allows you to control various extra settings used when submitting a command to the ledger. Each of these `meta` fields is optional.

Note: You cannot currently use `commandIds` anywhere else in the JSON API, but you can use it for observing the results of its commands outside the JSON API in logs or via the Ledger API's *Command Services*

```
{
  "templateId":
↪"a3b788b4dc18dc060bfb82366ae6dc055b1e361d646d5cfdb1b729607e344336:Iou:IouTransfer
↪",
  "payload": {
    "observers": [],
    "issuer": "Alice",
    "amount": "999.99",
    "currency": "USD",
    "owner": "Alice"
  },
  "meta": {
    "commandId": "a unique ID",
    "actAs": ["Alice"],
    "readAs": ["PublicParty"],
    "deduplicationPeriod": {
      "durationInMillis": 10000,
      "type": "Duration"
    },
    "submissionId": "d2f941b1-ee5c-4634-9a51-1335ce6902fa"
  }
}
```

Where:

`commandId` - optional field, a unique string identifying the command.
`actAs` - a non-empty list of parties, overriding the set from the JWT user; must be a subset of the JWT user's set.

`readAs` - a list of parties, overriding the set from the JWT user; must be a subset of the JWT user's set.
`submissionId` - a string, used for *deduplicating retried requests*. If you do not set it, a random one will be chosen, effectively treating the request as unique and disabling deduplication.
`deduplicationPeriod` - either a `Duration` as above, which is how far back in time prior commands will be searched for this submission, or an `Offset` as follows, which is the earliest ledger offset after which to search for the submission.

```
"deduplicationPeriod": {
  "offset": "0000000000000083",
  "type": "Offset"
}
```

Exercise by Contract ID

The JSON command below, demonstrates how to exercise an `Iou_Transfer` choice on an `Iou` contract:

```
    choice Iou_Transfer : ContractId IouTransfer
      with
        newOwner : Party
      controller owner
      do create IouTransfer with iou = this; newOwner
```

HTTP Request

URL: `/v1/exercise`
Method: `POST`
Content-Type: `application/json`
Content:

```
{
    "templateId":
↪"a3b788b4dc18dc060bfb82366ae6dc055b1e361d646d5cfdb1b729607e344336:Iou:IouTransfer
↪",
    "choiceInterfaceId":
↪"a3b788b4dc18dc060bfb82366ae6dc055b1e361d646d5cfdb1b729607e344336:Iou:IouTransferInterfac
↪",
    "contractId": "#124:0",
    "choice": "Iou_Transfer",
    "argument": {
        "newOwner": "Alice"
    }
}
```

Where:

`templateId` - contract template or interface identifier, same as in *create request*,
`choiceInterfaceId` - *optional* template or interface that defines the choice, same format as `templateId`,
`contractId` - contract identifier, the value from the *create response*,
`choice` - Daml contract choice, that is being exercised,

`argument` - contract choice argument(s).

`templateId` and `choiceInterfaceId` are treated as with *exercise by key*. However, because `contractId` is always unambiguous, you may alternatively simply specify the interface ID as the `templateId` argument, and ignore `choiceInterfaceId` entirely. This isn't true of exercise-by-key or create-and-exercise, so we suggest treating this request as if this alternative isn't available.

HTTP Response

Content-Type: `application/json`
Content:


```
{
    "status": 200,
    "result": {
        "exerciseResult": "#201:1",
        "events": [
            {
                "archived": {
                    "contractId": "#124:0",
                    "templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:Iou"
                }
            },
            {
                "created": {
                    "observers": [],
                    "agreementText": "",
                    "payload": {
                        "iou": {
                            "observers": [],
                            "issuer": "Alice",
                            "amount": "999.99",
                            "currency": "USD",
                            "owner": "Alice"
                        },
                        "newOwner": "Alice"
                    },
                    "signatories": [
                        "Alice"
                    ],
                    "contractId": "#201:1",
                    "templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:IouTransfer
↪"
                }
            }
        ],
        "completionOffset":"0000000000000083"
    }
}
```


Where:

`status` field matches the HTTP response status code returned in the HTTP header,
`result` field contains contract choice execution details:

- `exerciseResult` field contains the return value of the exercised contract choice.
- `events` contains an array of contracts that were archived and created as part of the choice execution. The array may contain: **zero or many** `{"archived": {...}}` and **zero or many** `{"created": {...}}` elements. The order of the contracts is the same as on the ledger.
- `completionOffset` is the ledger offset of the transaction containing the exercise's ledger changes.

Exercise by Contract Key

The JSON command below, demonstrates how to exercise the `Archive` choice on the `Account` contract with a `(Party, Text)` *contract key* defined like this:

```
template Account with
    owner : Party
    number : Text
    status : AccountStatus
  where
    signatory owner
    key (owner, number) : (Party, Text)
    maintainer key._1
```

HTTP Request

URL: `/v1/exercise`
Method: `POST`
Content-Type: `application/json`
Content:

```
{
    "templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Account:Account
↪",
    "key": {
        "_1": "Alice",
        "_2": "abc123"
    },
    "choiceInterfaceId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Account:AccountInterfac
↪",
    "choice": "Archive",
    "argument": {}
}
```

Where:

`templateId` - contract template identifier, same as in *create request*,
`key` - contract key, formatted according to the *Daml-LF JSON Encoding*,
`choiceInterfaceId` - *optional* template or interface that defines the choice, same format as `templateId`,
`choice` - Daml contract choice, that is being exercised,
`argument` - contract choice argument(s), empty, because `Archive` does not take any.

`key` is always searched in relation to the `templateId`. The `choice`, on the other hand, is searched according to `choiceInterfaceId`; if `choiceInterfaceId` is not specified, `templateId` is its default. We recommend always specifying `choiceInterfaceId` when invoking an interface choice; however, if the set of Daml-LF packages on the participant only contains one choice with a given name associated with `templateId`, that choice will be exercised, regardless of where it is defined. If a template *and* one or more of the interfaces it implements declares a choice, and `choiceInterfaceId` is not used, the one directly defined on the choice will be exercised. If choice selection is still ambiguous given these rules, the endpoint will fail as if the choice isn't defined.

HTTP Response

Formatted similar to *Exercise by Contract ID response*.

Create and Exercise in the Same Transaction

This command allows creating a contract and exercising a choice on the newly created contract in the same transaction.

HTTP Request

URL: `/v1/create-and-exercise`
Method: `POST`
Content-Type: `application/json`
Content:

```
{
  "templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:Iou",
  "payload": {
    "observers": [],
    "issuer": "Alice",
    "amount": "999.99",
    "currency": "USD",
    "owner": "Alice"
  },
  "choiceInterfaceId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:IouInterface
↪",
  "choice": "Iou_Transfer",
  "argument": {
    "newOwner": "Bob"
  }
}
```

Where:

`templateId` - the initial contract template identifier, in the same format as in the *create request*,
`payload` - the initial contract fields as defined in the Daml template and formatted according to *Daml-LF JSON Encoding*,

`choiceInterfaceId` - *optional* template or interface that defines the choice, same format as `templateId`,
`choice` - Daml contract choice, that is being exercised,
`argument` - contract choice argument(s).

`templateId` and `choiceInterfaceId` are treated as with *exercise by key*, with the exception that it is `payload`, not `key`, strictly interpreted according to `templateId`.

HTTP Response

Please note that the response below is for a consuming choice, so it contains:

`created` and `archived` events for the initial contract (`"contractId": "#1:0"`), which was created and archived right away when a consuming choice was exercised on it,
a `created` event for the contract that is the result of exercising the choice (`"contractId": "#1:2"`).
Content-Type: `application/json`
Content:

```
{
  "result": {
    "exerciseResult": "#1:2",
    "events": [
      {
        "created": {
          "observers": [],
          "agreementText": "",
          "payload": {
            "observers": [],
            "issuer": "Alice",
            "amount": "999.99",
            "currency": "USD",
            "owner": "Alice"
          },
          "signatories": [
            "Alice"
          ],
          "contractId": "#1:0",
          "templateId":
↪"a3b788b4dc18dc060bfb82366ae6dc055b1e361d646d5cfdb1b729607e344336:Iou:Iou"
        }
      },
      {
        "archived": {
          "contractId": "#1:0",
          "templateId":
↪"a3b788b4dc18dc060bfb82366ae6dc055b1e361d646d5cfdb1b729607e344336:Iou:Iou"
        }
      },
      {
        "created": {
          "observers": [
            "Bob"
          ],
          "agreementText": "",
```

(continues on next page)

(continued from previous page)


```
          "payload": {
            "iou": {
              "observers": [],
              "issuer": "Alice",
              "amount": "999.99",
              "currency": "USD",
              "owner": "Alice"
            },
            "newOwner": "Bob"
          },
          "signatories": [
            "Alice"
          ],
          "contractId": "#1:2",
          "templateId":
↪"a3b788b4dc18dc060bfb82366ae6dc055b1e361d646d5cfdb1b729607e344336:Iou:IouTransfer
↪"
        }
      }
    ]
  },
  "status": 200
}
```


Fetch Contract by Contract ID

HTTP Request

URL: `/v1/fetch`
Method: `POST`
Content-Type: `application/json`
Content:

application/json body:


```
{
  "contractId": "#201:1",
  "templateId":
↪"a3b788b4dc18dc060bfb82366ae6dc055b1e361d646d5cfdb1b729607e344336:Iou:IouTransfer
↪"
}
```


`readers` may be passed as with *Query*. `templateId` is optional, but you are strongly advised to always pass it explicitly to minimize the data read from the Ledger API to answer the query. It can be either a template ID or an interface ID.

Contract Not Found HTTP Response

Content-Type: `application/json`
Content:

```
{
    "status": 200,
    "result": null
}
```

Contract Found HTTP Response

Content-Type: `application/json`
Content:

```
{
    "status": 200,
    "result": {
        "observers": [],
        "agreementText": "",
        "payload": {
            "iou": {
                "observers": [],
                "issuer": "Alice",
                "amount": "999.99",
                "currency": "USD",
                "owner": "Alice"
            },
            "newOwner": "Alice"
        },
        "signatories": [
            "Alice"
        ],
        "contractId": "#201:1",
        "templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:IouTransfer
↪"
    }
}
```

Fetch Contract by Key

Show the currently active contract that matches a given key.

The websocket endpoint */v1/stream/fetch* can be used to search multiple keys in the same request, or in place of iteratively invoking this endpoint to respond to changes on the ledger.

HTTP Request

URL: `/v1/fetch`
Method: `POST`
Content-Type: `application/json`
Content:

```
{
    "templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Account:Account
↪",
    "key": {
        "_1": "Alice",
        "_2": "abc123"
    }
}
```

`readers` may be passed as with *Query*.

Contract Not Found HTTP Response

Content-Type: `application/json`
Content:

```
{
    "status": 200,
    "result": null
}
```

Contract Found HTTP Response

Content-Type: `application/json`
Content:

```
{
    "status": 200,
    "result": {
        "observers": [],
        "agreementText": "",
        "payload": {
            "owner": "Alice",
            "number": "abc123",
            "status": {
                "tag": "Enabled",
                "value": "2020-01-01T00:00:01Z"
            }
        },
        "signatories": [
            "Alice"
        ],
        "key": {
            "_1": "Alice",
```

(continues on next page)

(continued from previous page)

```
            "_2": "abc123"
        },
        "contractId": "#697:0",
        "templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Account:Account
↪"
    }
}
```

Get All Active Contracts

List all currently active contracts for all known templates.

Note: Retrieved contracts do not get persisted into a query store database. Query store is a search index and can be used to optimize search latency. See *Start HTTP service* for information on how to start JSON API service with a query store enabled.

Note: You can only query active contracts with the `/v1/query` endpoint. Archived contracts (those that were archived or consumed during an exercise operation) will not be shown in the results.

HTTP Request

URL: `/v1/query`
Method: `GET`
Content: <EMPTY>

HTTP Response

The response is the same as for the POST method below.

Get All Active Contracts Matching a Given Query

List currently active contracts that match a given query.

The websocket endpoint */v1/stream/query* can be used in place of iteratively invoking this endpoint to respond to changes on the ledger.

HTTP Request

URL: `/v1/query`
Method: `POST`
Content-Type: `application/json`
Content:

```
{
    "templateIds": [
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:Iou"],
    "query": {"amount": 999.99},
    "readers": ["Alice"]
}
```

Where:

`templateIds` – either an array of contract template identifiers or an array containing a single interface identifier to search through. Mixing of template ID's and interface ID's, or specifying more than one interface ID is not allowed.
`query` – search criteria to apply to the specified `templateIds`, formatted according to the *Query Language*.
`readers` – *optional* non-empty list of parties to query as; must be a subset of the actAs/readAs parties in the JWT

Empty HTTP Response

Content-Type: `application/json`
Content:

```
{
    "status": 200,
    "result": []
}
```

Nonempty HTTP Response

Content-Type: `application/json`
Content:

```
{
    "result": [
        {
            "observers": [],
            "agreementText": "",
            "payload": {
                "observers": [],
                "issuer": "Alice",
                "amount": "999.99",
                "currency": "USD",
                "owner": "Alice"
            },
            "signatories": [
```

(continues on next page)

(continued from previous page)

```
                "Alice"
            ],
            "contractId": "#52:0",
            "templateId":
↪"b10d22d6c2f2fae41b353315cf893ed66996ecb0abe4424ea6a81576918f658a:Iou:Iou"
        }
    ],
    "status": 200
}
```

Where

`result` contains an array of contracts, each contract formatted according to *Daml-LF JSON Encoding*,
`status` matches the HTTP status code returned in the HTTP header.

Nonempty HTTP Response With Unknown Template IDs Warning

Content-Type: `application/json`
Content:

```
{
    "warnings": {
        "unknownTemplateIds": ["UnknownModule:UnknownEntity"]
    },
    "result": [
        {
            "observers": [],
            "agreementText": "",
            "payload": {
                "observers": [],
                "issuer": "Alice",
                "amount": "999.99",
                "currency": "USD",
                "owner": "Alice"
            },
            "signatories": [
                "Alice"
            ],
            "contractId": "#52:0",
            "templateId":
↪"b10d22d6c2f2fae41b353315cf893ed66996ecb0abe4424ea6a81576918f658a:Iou:Iou"
        }
    ],
    "status": 200
}
```

Fetch Parties by Identifiers

URL: `/v1/parties`
Method: `POST`
Content-Type: `application/json`
Content:

```
["Alice", "Bob", "Dave"]
```

If an empty JSON array is passed: `[]`, this endpoint returns BadRequest(400) error:

```
{
  "status": 400,
  "errors": [
    "JsonReaderError. Cannot read JSON: <[]>. Cause: spray.json.
↪DeserializationException: must be a list with at least 1 element"
  ]
}
```

HTTP Response

Content-Type: `application/json`
Content:

```
{
  "status": 200,
  "result": [
    {
      "identifier": "Alice",
      "displayName": "Alice & Co. LLC",
      "isLocal": true
    },
    {
      "identifier": "Bob",
      "displayName": "Bob & Co. LLC",
      "isLocal": true
    },
    {
      "identifier": "Dave",
      "isLocal": true
    }
  ]
}
```

Please note that the order of the party objects in the response is not guaranteed to match the order of the passed party identifiers.

Where

`identifier` - a stable unique identifier of a Daml party,
`displayName` - optional human readable name associated with the party. Might not be unique,
`isLocal` - true if party is hosted by the backing participant.

Response With Unknown Parties Warning

Content-Type: `application/json`
Content:

```
{
  "result": [
    {
      "identifier": "Alice",
      "displayName": "Alice & Co. LLC",
      "isLocal": true
    }
  ],
  "warnings": {
    "unknownParties": ["Erin"]
  },
  "status": 200
}
```

The `result` might be an empty JSON array if none of the requested parties is known.

Fetch All Known Parties

URL: `/v1/parties`
Method: `GET`
Content: <EMPTY>

HTTP Response

The response is the same as for the POST method above.

Allocate a New Party

This endpoint is a JSON API proxy for the Ledger API's *AllocatePartyRequest*. For more information about party management, please refer to *Provisioning Identifiers* part of the Ledger API documentation.

HTTP Request

URL: `/v1/parties/allocate`
Method: `POST`
Content-Type: `application/json`
Content:

```
{
  "identifierHint": "Carol",
  "displayName": "Carol & Co. LLC"
}
```

Please refer to *AllocateParty* documentation for information about the meaning of the fields.

All fields in the request are optional, this means that an empty JSON object is a valid request to allocate a new party:

```
{}
```

HTTP Response

```
{
  "result": {
    "identifier": "Carol",
    "displayName": "Carol & Co. LLC",
    "isLocal": true
  },
  "status": 200
}
```

Create a New User

This endpoint exposes the Ledger API's *CreateUser RPC*.

HTTP Request

URL: `/v1/user/create`
Method: `POST`
Content-Type: `application/json`
Content:

```
{
  "userId": "carol",
  "primaryParty": "Carol",
  "rights": [
    {
      "type": "CanActAs",
      "party": "Carol"
    },
    {
      "type": "CanReadAs",
      "party": "Alice"
    },
    {
      "type": "CanReadAs",
      "party": "Bob"
    },
    {
      "type": "ParticipantAdmin"
    }
  ]
}
```

Please refer to *CreateUser RPC* documentation for information about the meaning of the fields.

Only the userId fields in the request is required, this means that an JSON object containing only it is a valid request to create a new user.

HTTP Response

```
{
  "result": {},
  "status": 200
}
```

Get Authenticated User Information

This endpoint exposes the Ledger API's *GetUser RPC*.

The user ID will always be filled out with the user specified via the currently used user token.

HTTP Request

URL: `/v1/user`
Method: `GET`

HTTP Response

```
{
  "result": {
    "userId": "carol",
    "primaryParty": "Carol"
  },
  "status": 200
}
```

Get Specific User Information

This endpoint exposes the Ledger API's *GetUser RPC*.

HTTP Request

URL: `/v1/user`
Method: `POST`
Content-Type: `application/json`
Content:

```
{
  "userId": "carol"
}
```

Please refer to *GetUser RPC* documentation for information about the meaning of the fields.

HTTP Response

```
{
  "result": {
    "userId": "carol",
    "primaryParty": "Carol"
  },
  "status": 200
}
```

Delete Specific User

This endpoint exposes the Ledger API's *DeleteUser RPC*.

HTTP Request

URL: `/v1/user/delete`
Method: `POST`
Content-Type: `application/json`
Content:

```
{
  "userId": "carol"
}
```

Please refer to *DeleteUser RPC* documentation for information about the meaning of the fields.

HTTP Response

```
{
  "result": {},
  "status": 200
}
```

List Users

This endpoint exposes the Ledger API's *ListUsers RPC*.

HTTP Request

URL: `/v1/users`
Method: `GET`

HTTP Response

```
{
  "result": [
    {
        "userId": "carol",
        "primaryParty": "Carol"
    },
    {
        "userId": "bob",
        "primaryParty": "Bob"
    }
  ],
  "status": 200
}
```

Grant User Rights

This endpoint exposes the Ledger API's *GrantUserRights RPC*.

HTTP Request

URL: `/v1/user/rights/grant`
Method: `POST`
Content-Type: `application/json`
Content:

```
{
  "userId": "carol",
  "rights": [
    {
      "type": "CanActAs",
      "party": "Carol"
    },
    {
      "type": "CanReadAs",
      "party": "Alice"
    },
    {
      "type": "CanReadAs",
      "party": "Bob"
    },
    {
      "type": "ParticipantAdmin"
    }
```

(continues on next page)

(continued from previous page)

```
  ]
}
```

Please refer to *GrantUserRights RPC* documentation for information about the meaning of the fields.

HTTP Response

```
{
  "result": [
    {
      "type": "CanActAs",
      "party": "Carol"
    },
    {
      "type": "CanReadAs",
      "party": "Alice"
    },
    {
      "type": "CanReadAs",
      "party": "Bob"
    },
    {
      "type": "ParticipantAdmin"
    }
  ],
  "status": 200
}
```

Returns the rights that were newly granted.

Revoke User Rights

This endpoint exposes the Ledger API's *RevokeUserRights RPC*.

HTTP Request

URL: `/v1/user/rights/revoke`
Method: `POST`
Content-Type: `application/json`
Content:

```
{
  "userId": "carol",
  "rights": [
    {
      "type": "CanActAs",
      "party": "Carol"
    },
    {
      "type": "CanReadAs",
```

(continues on next page)

(continued from previous page)

```
      "party": "Alice"
    },
    {
      "type": "CanReadAs",
      "party": "Bob"
    },
    {
      "type": "ParticipantAdmin"
    }
  ]
}
```

Please refer to *RevokeUserRights RPC* documentation for information about the meaning of the fields.

HTTP Response

```
{
  "result": [
    {
      "type": "CanActAs",
      "party": "Carol"
    },
    {
      "type": "CanReadAs",
      "party": "Alice"
    },
    {
      "type": "CanReadAs",
      "party": "Bob"
    },
    {
      "type": "ParticipantAdmin"
    }
  ],
  "status": 200
}
```

Returns the rights that were actually granted.

List Authenticated User Rights

This endpoint exposes the Ledger API's *ListUserRights RPC*.

The user ID will always be filled out with the user specified via the currently used user token.

HTTP Request

URL: `/v1/user/rights`
Method: `GET`

HTTP Response

```
{
  "result": [
    {
      "type": "CanActAs",
      "party": "Carol"
    },
    {
      "type": "CanReadAs",
      "party": "Alice"
    },
    {
      "type": "CanReadAs",
      "party": "Bob"
    },
    {
      "type": "ParticipantAdmin"
    }
  ],
  "status": 200
}
```

List Specific User Rights

This endpoint exposes the Ledger API's *ListUserRights RPC*.

HTTP Request

URL: `/v1/user/rights`
Method: `POST`
Content-Type: `application/json`
Content:

```
{
  "userId": "carol"
}
```

Please refer to *ListUserRights RPC* documentation for information about the meaning of the fields.

HTTP Response


```
{
  "result": [
    {
      "type": "CanActAs",
      "party": "Carol"
    },
    {
      "type": "CanReadAs",
      "party": "Alice"
    },
    {
      "type": "CanReadAs",
      "party": "Bob"
    },
    {
      "type": "ParticipantAdmin"
    }
  ],
  "status": 200
}
```


List All DALF Packages

HTTP Request

URL: `/v1/packages`
Method: `GET`
Content: <EMPTY>

HTTP Response


```
{
  "result": [
    "c1f1f00558799eec139fb4f4c76f95fb52fa1837a5dd29600baa1c8ed1bdccfd",
    "733e38d36a2759688a4b2c4cec69d48e7b55ecc8dedc8067b815926c917a182a",
    "bfcd37bd6b84768e86e432f5f6c33e25d9e7724a9d42e33875ff74f6348e733f",
    "40f452260bef3f29dede136108fc08a88d5a5250310281067087da6f0baddff7",
    "8a7806365bbd98d88b4c13832ebfa305f6abaeaf32cfa2b7dd25c4fa489b79fb"
  ],
  "status": 200
}
```


Where `result` is the JSON array containing the package IDs of all loaded DALFs.

Download a DALF Package

HTTP Request

URL: `/v1/packages/<package ID>`
Method: `GET`
Content: <EMPTY>

Note that the desired package ID is specified in the URL.

HTTP Response, status: 200 OK

Transfer-Encoding: `chunked`
Content-Type: `application/octet-stream`
Content: <DALF bytes>

The content (body) of the HTTP response contains raw DALF package bytes, without any encoding. Note that the package ID specified in the URL is actually the SHA-256 hash of the downloaded DALF package and can be used to validate the integrity of the downloaded content.

HTTP Response With Error, Any Status Different from 200 OK

Any status different from `200 OK` will be in the format specified below.

Content-Type: `application/json`
Content:

```
{
    "errors": [
        "io.grpc.StatusRuntimeException: NOT_FOUND"
    ],
    "status": 500
}
```

Upload a DAR File

HTTP Request

URL: `/v1/packages`
Method: `POST`
Content-Type: `application/octet-stream`
Content: <DAR bytes>

The content (body) of the HTTP request contains raw DAR file bytes, without any encoding.

HTTP Response, Status: 200 OK

Content-Type: `application/json`
Content:

```
{
    "result": 1,
    "status": 200
}
```

HTTP Response With Error

Content-Type: `application/json`
Content:

```
{
    "errors": [
        "io.grpc.StatusRuntimeException: INVALID_ARGUMENT: Invalid argument:
↪Invalid DAR: package-upload, content: [}]"
    ],
    "status": 500
}
```

Metering Report

For a description of participant metering, the parameters, and the report format see the *Participant Metering*.

URL: `/v1/metering-report`
Method: `POST`
Content-Type: `application/json`
Content:

```
{
  "from": "2022-01-01",
  "to": "2022-02-01",
  "application": "some-application"
}
```

HTTP Response

Content-Type: `application/json`
Content:

```
{
  "status": 200,
  "result": {
    "participant": "some-participant",
    "request": {
      "from": "2022-01-01T00:00:00Z",
```

(continues on next page)

(continued from previous page)

```
      "to": "2022-02-01T00:00:00Z"
    },
    "final": true,
    "applications": [
      {
        "application": "some-application",
        "events": 42
      }
    ]
  }
}
```

Streaming API

Two subprotocols must be passed with every request, as described in *Auth via WebSockets*.

JavaScript/Node.js example demonstrating how to establish Streaming API connection:

```
const wsProtocol = "daml.ws.auth";
const tokenPrefix = "jwt.token.";
const jwt =
  "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJodHRwczovL2RhbWwuY29tL2xlZGdlci1hcGkiOnsibGVkZ2VySWQiOiJNeUxlZGdlciIsImFwcGxpY2F0aW9uS
↪34zzF_fbWv7p60r5s1kKzwndvGdsJDX-W4Xhm4oVdp";
const subprotocols = [`${tokenPrefix}${jwt}`, wsProtocol];

const ws = new WebSocket("ws://localhost:7575/v1/stream/query", subprotocols);

ws.addEventListener("open", function open() {
  ws.send(JSON.stringify({templateIds: [
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:Iou"]}));
});

ws.addEventListener("message", function incoming(data) {
  console.log(data);
});
```

Please note that Streaming API does not allow multiple requests over the same WebSocket connection. The server returns an error and disconnects if second request received over the same WebSocket connection.

Error and Warning Reporting

Errors and warnings reported as part of the regular `on-message` flow: `ws.addEventListener("message", ...)`.

Streaming API error messages formatted the same way as *synchronous API errors*.

Streaming API reports only one type of warnings - unknown template IDs, which is formatted as:

```
{"warnings":{"unknownTemplateIds":<JSON Array of template ID strings>>}}
```

Error and Warning Examples

```
{"warnings": {"unknownTemplateIds": ["UnknownModule:UnknownEntity"]}}

{
  "errors":["JsonReaderError. Cannot read JSON: <{\"templateIds\":[]}>. Cause:
↪spray.json.DeserializationException: search requires at least one item in
↪'templateIds'"],
  "status":400
}

{
  "errors":["Multiple requests over the same WebSocket connection are not allowed.
↪"],
  "status":400
}

{
  "errors":["Could not resolve any template ID from request."],
  "status":400
}
```

Contracts Query Stream

URL: `/v1/stream/query`
Scheme: `ws`
Protocol: `WebSocket`

List currently active contracts that match a given query, with continuous updates.

Simpler use-cases that do not require continuous updates should use the simpler */v1/query* endpoint instead.

`application/json` body must be sent first, formatted according to the *Query Language*:

```
{"templateIds": [
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:Iou"]}
```

Multiple queries may be specified in an array, for overlapping or different sets of template IDs.:

```
[
    {"templateIds": [
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:Iou"],
↪"query": {"amount": {"%lte": 50}}},
    {"templateIds": [
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:OtherIou:OtherIou
↪"], "query": {"amount": {"%gt": 50}}},
    {"templateIds": [
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:Iou"]}
]
```

Only one interface ID can be provided in `templateIds`. An interface ID can be used in all queries:

```
[
    {"templateIds": [
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Ifc:Ifc"],
↪"query": {"amount": {"%lte": 50}}},
    {"templateIds": [
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Ifc:Ifc"],
↪"query": {"amount": {"%gt": 50}}},
    {"templateIds": [
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Ifc:Ifc"]}
]
```

Mixing of template ID's and interface ID's or specifying more than one interface ID across queries is not allowed. BadRequest(400) error will be returned.:

```
[
    {"templateIds": [
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:Iou"],
↪"query": {"amount": {"%lte": 50}}},
    {"templateIds": [
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Ifc:Ifc"],
↪"query": {"amount": {"%gt": 50}}},
    {"templateIds": [
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Ifc:Ifc"]}
]
```

Queries have two ways to specify an offset.

An `offset`, a string supplied by an earlier query output message, may optionally be specified alongside each query itself:

```
[
    {"templateIds": [
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:Iou"],
↪"query": {"amount": {"%lte": 50}}},
    {"templateIds": [
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:Iou"],
↪"query": {"amount": {"%gt": 50}}},
    {"templateIds": [
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:Iou"],
↪"offset": "5609"}
]
```

If specified, the stream will include only contract creations and archivals *after* the response body that included that offset. Queries with no offset will begin with all active contracts for that query, as usual.

If an offset is specified *before* the queries, as a separate body, it will be used as a default offset for all queries that do not include an offset themselves:

```
{"offset": "4307"}
```

For example, if this message preceded the above 3-query example, it would be as if `"4307"` had been specified for the first two queries, while `"5609"` would be used for the third query.

If any offset has been pruned, the websocket will immediately fail with code 1011 and message `internal error`.

The output is a series of JSON documents, each `payload` formatted according to *Daml-LF JSON Encoding*:

```
{
    "events": [{
        "created": {
            "observers": [],
            "agreementText": "",
            "payload": {
                "observers": [],
                "issuer": "Alice",
                "amount": "999.99",
                "currency": "USD",
                "owner": "Alice"
            },
            "signatories": ["Alice"],
            "contractId": "#1:0",
            "templateId":
↪"eb3b150383a979d6765b8570a17dd24ae8d8b63418ee5fd20df20ad2a1c13976:Iou:Iou"
        },
        "matchedQueries": [1, 2]
    }]
}
```

where `matchedQueries` indicates the 0-based indices into the request list of queries that matched this contract.

Every `events` block following the end of contracts that existed when the request started includes an `offset`. The stream is guaranteed to send an offset immediately at the beginning of this live data, which may or may not contain any `events`; if it does not contain events and no events were emitted before, it may be `null` if there was no transaction on the ledger or a string representing the current ledger end; otherwise, it will be a string. For example, you might use it to turn off an initial loading indicator:

```
{
    "events": [],
    "offset": "2"
}
```

Note: Events in the following live data may include `events` that precede this `offset` if an earlier per-query `offset` was specified.

This has been done with the intent of allowing to use per-query `offset` s to efficiently use a single connection to multiplex various requests. To give an example of how this would work, let's say that there are two contract templates, `A` and `B` . Your application first queries for `A` s without specifying an offset. Then some client-side interaction requires the application to do the same for `B` s. The application can save the latest observed offset for the previous query, which let's say is `42`, and issue a new request that queries for all `B` s without specifying an offset and all `A` s from `42`. While this happens on the client, a few more `A` s and `B` s are created and the new request is issued once the latest offset is `47`. The response to this will contain a message with all active `B` s, followed by the message reporting the offset `47`, followed by a stream of live updates that contains new `A` s starting from `42` and new `B` s starting from `47` .

To keep the stream alive, you'll occasionally see messages like this, which can be safely ignored if

you do not need to capture the last seen ledger offset:

```
{"events":[],"offset":"5609"}
```

where `offset` is the last seen ledger offset.

After submitting an `Iou_Split` exercise, which creates two contracts and archives the one above, the same stream will eventually produce:

```
{
    "events": [{
        "archived": {
            "contractId": "#1:0",
            "templateId":
↪"eb3b150383a979d6765b8570a17dd24ae8d8b63418ee5fd20df20ad2a1c13976:Iou:Iou"
        }
    }, {
        "created": {
            "observers": [],
            "agreementText": "",
            "payload": {
                "observers": [],
                "issuer": "Alice",
                "amount": "42.42",
                "currency": "USD",
                "owner": "Alice"
            },
            "signatories": ["Alice"],
            "contractId": "#2:1",
            "templateId":
↪"eb3b150383a979d6765b8570a17dd24ae8d8b63418ee5fd20df20ad2a1c13976:Iou:Iou"
        },
        "matchedQueries": [0, 2]
    }, {
        "created": {
            "observers": [],
            "agreementText": "",
            "payload": {
                "observers": [],
                "issuer": "Alice",
                "amount": "957.57",
                "currency": "USD",
                "owner": "Alice"
            },
            "signatories": ["Alice"],
            "contractId": "#2:2",
            "templateId":
↪"eb3b150383a979d6765b8570a17dd24ae8d8b63418ee5fd20df20ad2a1c13976:Iou:Iou"
        },
        "matchedQueries": [1, 2]
    }],
    "offset": "3"
}
```

If any template IDs are found not to resolve, the first element of the stream will report them:

```
{"warnings": {"unknownTemplateIds": ["UnknownModule:UnknownEntity"]}}
```

and the stream will continue, provided that at least one template ID resolved properly.

Aside from "created" and "archived" elements, "error" elements may appear, which contain a string describing the error. The stream will continue in these cases, rather than terminating.

Some notes on behavior:

1. Each result array means this is what would have changed if you just polled /v1/query iteratively. In particular, just as polling search can miss contracts (as a create and archive can be paired between polls), such contracts may or may not appear in any result object.
2. No archived ever contains a contract ID occurring within a created in the same array. So, for example, supposing you are keeping an internal map of active contracts keyed by contract ID, you can apply the created first or the archived first, forwards, backwards, or in random order, and be guaranteed to get the same results.
3. Within a given array, if an archived and created refer to contracts with the same template ID and *contract key*, the archived is guaranteed to occur before the created.
4. Except in cases of #3, within a single response array, the order of created and archived is undefined and does not imply that any element occurred before or after any other one.
5. You will almost certainly receive contract IDs in archived that you never received a created for. These are contracts that query filtered out, but for which the server no longer is aware of that. You can safely ignore these. However, such phantom archives *are* guaranteed to represent an actual archival *on the ledger*, so if you are keeping a more global dataset outside the context of this specific search, you can use that archival information as you wish.

Fetch by Key Contracts Stream

URL: /v1/stream/fetch
Scheme: ws
Protocol: WebSocket

List currently active contracts that match one of the given {templateId, key} pairs, with continuous updates.

Simpler use-cases that search for only a single key and do not require continuous updates should use the simpler */v1/fetch* endpoint instead.

application/json body must be sent first, formatted according to the following rule:

```
[
    {"templateId": "<template ID 1>", "key": <key 1>},
    {"templateId": "<template ID 2>", "key": <key 2>},
    ...
    {"templateId": "<template ID N>", "key": <key N>}
]
```

Where:

templateId – contract template identifier, same as in *create request*,
key – contract key, formatted according to the *Daml-LF JSON Encoding*,

Example:

```
[
    {"templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Account:Account
↪", "key": {"_1": "Alice", "_2": "abc123"}},
```

(continues on next page)

(continued from previous page)

```
    {"templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Account:Account
↪", "key": {"_1": "Alice", "_2": "def345"}}
]
```

The output stream has the same format as the output from the *Contracts Query Stream*. We further guarantee that for every `archived` event appearing on the stream there has been a matching `created` event earlier in the stream, except in the case of missing `contractIdAtOffset` fields in the case described below.

You may supply optional `offset` s for the stream, exactly as with query streams. However, you should supply with each `{templateId, key}` pair a `contractIdAtOffset`, which is the contract ID currently associated with that pair at the point of the given offset, or `null` if no contract ID was associated with the pair at that offset. For example, with the above keys, if you had one `"abc123"` contract but no `"def345"` contract, you might specify:

```
[
    {"templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Account:Account
↪", "key": {"_1": "Alice", "_2": "abc123"},
     "contractIdAtOffset": "#1:0"},
    {"templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Account:Account
↪", "key": {"_1": "Alice", "_2": "def345"},
     "contractIdAtOffset": null}
]
```

If every `contractIdAtOffset` is specified, as is so in the example above, you will not receive any `archived` events for contracts created before the offset *unless* those contracts are identified in a `contractIdAtOffset`. By contrast, if any `contractIdAtOffset` is missing, `archived` event filtering will be disabled, and you will receive phantom archives as with query streams.

Healthcheck Endpoints

The HTTP JSON API provides two healthcheck endpoints for integration with schedulers like Kubernetes.

Liveness Check

URL: `/livez`
Method: `GET`

A status code of `200` indicates a successful liveness check.

This is an unauthenticated endpoint intended to be used as a liveness probe.

Readiness Check

URL: `/readyz`
Method: `GET`

A status code of `200` indicates a successful readiness check.

This is an unauthenticated endpoint intended to be used as a readiness probe. It validates both the ledger connection as well as the database connection.

1.12.4.2 Daml-LF JSON Encoding

We describe how to decode and encode Daml-LF values as JSON. For each Daml-LF type we explain what JSON inputs we accept (decoding), and what JSON output we produce (encoding).

If you use *the JavaScript code generator* with TypeScript, the generated types for templates and choices will incorporate the following automatically. You can use this to observe how these rules apply to your templates, or ignore this document and rely on the TypeScript type checker to tell you how to encode data for JSON API correctly.

Codec Library

At the library level, the output format is parameterized by two flags:

```
encodeDecimalAsString: boolean
encodeInt64AsString: boolean
```

The suggested defaults for both of these flags is false. If the intended recipient is written in JavaScript, however, note that the JavaScript data model will decode these as numbers, discarding data in some cases; encode-as-String avoids this, as mentioned with respect to `JSON.parse` below. **For that reason, the HTTP JSON API Service uses ``true`` for both flags.**

Type-directed Parsing

Note that throughout the document the decoding is type-directed. In other words, the same JSON value can correspond to many Daml-LF values, and a single Daml-LF value can correspond to multiple JSON encodings. This means it is crucial to know the expected type of a JSON-encoded LF value to make sense of it.

For that reason, you should parse the data into appropriate data types (including parsing numbers into appropriate representations) before doing any meaningful manipulations (e.g. comparison for equality).

ContractId

Contract ids are expressed as their string representation:

```
"123"
"XYZ"
"foo:bar#baz"
```

Decimal

Input

Decimals can be expressed as JSON numbers or as JSON strings. JSON strings are accepted using the same format that JSON accepts, and treated them as the equivalent JSON number:

```
-?(?:0|[1-9]\d*)(?:\.\d+)?(?:[eE][+-]?\d+)?
```

Note that JSON numbers would be enough to represent all Decimals. However, we also accept strings because in many languages (most notably JavaScript) use IEEE Doubles to express JSON numbers, and IEEE Doubles cannot express Daml-LF Decimals correctly. Therefore, we also accept strings so that JavaScript users can use them to specify Decimals that do not fit in IEEE Doubles.

Numbers must be within the bounds of Decimal, [-(10 -1) 10 , (10 -1) 10]. Numbers outside those bounds will be rejected. Numbers inside the bounds will always be accepted, using banker's rounding to fit them within the precision supported by Decimal.

A few valid examples:

```
42 --> 42
42.0 --> 42
"42" --> 42
9999999999999999999999999999.9999999999 -->
    9999999999999999999999999999.9999999999
-42 --> -42
"-42" --> -42
0 --> 0
-0 --> 0
0.30000000000000004 --> 0.3
2e3 --> 2000
```

A few invalid examples:

```
"  42  "
"blah"
99999999999999999999999999990
+42
```

Output

If encodeDecimalAsString is set, decimals are encoded as strings, using the format `-?[0-9]{1,28}(\.[0-9]{1,10})?`. If encodeDecimalAsString is not set, they are encoded as JSON numbers, also using the format `-?[0-9]{1,28}(\.[0-9]{1,10})?`.

Note that the flag encodeDecimalAsString is useful because it lets JavaScript consumers consume Decimals safely with the standard JSON.parse.

Int64

Input

Int64, much like Decimal, can be represented as JSON numbers and as strings, with the string representation being `[+-]?[0-9]+`. The numbers must fall within [-9223372036854775808, 9223372036854775807]. Moreover, if represented as JSON numbers, they must have no fractional part.

A few valid examples:

```
42
"+42"
-42
0
-0
9223372036854775807
"9223372036854775807"
-9223372036854775808
"-9223372036854775808"
```

A few invalid examples:

```
42.3
+42
9223372036854775808
-9223372036854775809
"garbage"
"   42 "
```

Output

If encodeInt64AsString is set, Int64s are encoded as strings, using the format `-?[0-9]+`. If encodeInt64AsString is not set, they are encoded as JSON numbers, also using the format `-?[0-9]+`.

Note that the flag encodeInt64AsString is useful because it lets JavaScript consumers consume Int64s safely with the standard `JSON.parse`.

Timestamp

Input

Timestamps are represented as ISO 8601 strings, rendered using the format `yyyy-mm-ddThh:mm:ss.ssssssZ`:

```
1990-11-09T04:30:23.123456Z
9999-12-31T23:59:59.999999Z
```

Parsing is a little bit more flexible and uses the format `yyyy-mm-ddThh:mm:ss(\.s+)?Z`, i.e. it's OK to omit the microsecond part partially or entirely, or have more than 6 decimals. Sub-second data beyond microseconds will be dropped. The UTC timezone designator must be included. The rationale behind the inclusion of the timezone designator is minimizing the risk that users pass in local times. Valid examples:

```
1990-11-09T04:30:23.1234569Z
1990-11-09T04:30:23Z
1990-11-09T04:30:23.123Z
0001-01-01T00:00:00Z
9999-12-31T23:59:59.999999Z
```

The timestamp must be between the bounds specified by Daml-LF and ISO 8601, [0001-01-01T00:00:00Z, 9999-12-31T23:59:59.999999Z].

JavaScript

```
> new Date().toISOString()
'2019-06-18T08:59:34.191Z'
```

Python

```
>>> datetime.datetime.utcnow().isoformat() + 'Z'
'2019-06-18T08:59:08.392764Z'
```

Java

```
import java.time.Instant;
class Main {
    public static void main(String[] args) {
        Instant instant = Instant.now();
        // prints 2019-06-18T09:02:16.652Z
        System.out.println(instant.toString());
    }
}
```

Output

Timestamps are encoded as ISO 8601 strings, rendered using the format `yyyy-mm-ddThh:mm:ss[.ssssss]Z`.

The sub-second part will be formatted as follows:

> If no sub-second part is present in the timestamp (i.e. the timestamp represents whole seconds), the sub-second part will be omitted entirely;
> If the sub-second part does not go beyond milliseconds, the sub-second part will be up to milliseconds, padding with trailing 0s if necessary;
> Otherwise, the sub-second part will be up to microseconds, padding with trailing 0s if necessary.

In other words, the encoded timestamp will either have no sub-second part, a sub-second part of length 3, or a sub-second part of length 6.

Party

Represented using their string representation, without any additional quotes:

```
"Alice"
"Bob"
```

Unit

Represented as empty object `{}`. Note that in JavaScript `{} !== {}`; however, `null` would be ambiguous; for the type `Optional Unit`, `null` decodes to `None`, but `{}` decodes to `Some ()`.

Additionally, we think that this is the least confusing encoding for Unit since unit is conceptually an empty record. We do not want to imply that Unit is used similarly to null in JavaScript or None in Python.

Date

Represented as an ISO 8601 date rendered using the format `yyyy-mm-dd`:

```
2019-06-18
9999-12-31
0001-01-01
```

The dates must be between the bounds specified by Daml-LF and ISO 8601, [0001-01-01, 9999-12-31].

Text

Represented as strings.

Bool

Represented as booleans.

Record

Input

Records can be represented in two ways. As objects:

```
{ f□: v□, ..., f□: v□ }
```

And as arrays:

```
[ v□, ..., v□ ]
```

Note that Daml-LF record fields are ordered. So if we have

```
record Foo = {f1: Int64, f2: Bool}
```

when representing the record as an array the user must specify the fields in order:

```
[42, true]
```

The motivation for the array format for records is to allow specifying tuple types closer to what it looks like in Daml. Note that a Daml tuple, i.e. (42, True), will be compiled to a Daml-LF record `Tuple2 { _1 = 42, _2 = True }`.

Output

Records are always encoded as objects.

List

Lists are represented as

```
[v□, ..., v□]
```

TextMap

TextMaps are represented as objects:

```
{ k□: v□, ..., k□: v□ }
```

GenMap

GenMaps are represented as lists of pairs:

```
[ [k□, v□], [k□, v□] ]
```

Order does not matter. However, any duplicate keys will cause the map to be treated as invalid.

Optional

Input

Optionals are encoded using `null` if the value is None, and with the value itself if it's Some. However, this alone does not let us encode nested optionals unambiguously. Therefore, nested Optionals are encoded using an empty list for None, and a list with one element for Some. Note that after the top-level Optional, all the nested ones must be represented using the list notation.

A few examples, using the form

```
JSON  -->  Daml-LF  :  Expected Daml-LF type
```

to make clear what the target Daml-LF type is:

```
null     -->  None                  : Optional Int64
null     -->  None                  : Optional (Optional Int64)
42       -->  Some 42               : Optional Int64
[]       -->  Some None             : Optional (Optional Int64)
[42]     -->  Some (Some 42)        : Optional (Optional Int64)
[[]]     -->  Some (Some None)      : Optional (Optional (Optional Int64))
[[42]]   -->  Some (Some (Some 42)) : Optional (Optional (Optional Int64))
...
```

Finally, if Optional values appear in records, they can be omitted to represent None. Given Daml-LF types

```
record Depth1 = { foo: Optional Int64 }
record Depth2 = { foo: Optional (Optional Int64) }
```

We have

```
{ }              -->  Depth1 { foo: None }            :  Depth1
{ }              -->  Depth2 { foo: None }            :  Depth2
{ foo: 42 }      -->  Depth1 { foo: Some 42 }         :  Depth1
{ foo: [42] }    -->  Depth2 { foo: Some (Some 42) }  :  Depth2
{ foo: null }    -->  Depth1 { foo: None }            :  Depth1
```

(continues on next page)

(continued from previous page)

```
{ foo: null }     -->  Depth2 { foo: None }            :  Depth2
{ foo: [] }       -->  Depth2 { foo: Some None }       :  Depth2
```

Note that the shortcut for records and Optional fields does not apply to Map (which are also represented as objects), since Map relies on absence of key to determine what keys are present in the Map to begin with. Nor does it apply to the `[f□, ..., f□]` record form; `Depth1 None` in the array notation must be written as `[null]`.

Type variables may appear in the Daml-LF language, but are always resolved before deciding on a JSON encoding. So, for example, even though `Oa` doesn't appear to contain a nested `Optional`, it may contain a nested `Optional` by virtue of substituting the type variable `a`:

```
record Oa a = { foo: Optional a }

{ foo: 42 }     -->  Oa { foo: Some 42 }        : Oa Int
{ }             -->  Oa { foo: None }           : Oa Int
{ foo: [] }     -->  Oa { foo: Some None }      : Oa (Optional Int)
{ foo: [42] }   -->  Oa { foo: Some (Some 42) } : Oa (Optional Int)
```

In other words, the correct JSON encoding for any LF value is the one you get when you have eliminated all type variables.

Output

Encoded as described above, never applying the shortcut for None record fields; e.g. `{ foo: None }` will always encode as `{ foo: null }`.

Variant

Variants are expressed as

```
{ tag: constructor, value: argument }
```

For example, if we have

```
variant Foo = Bar Int64 | Baz Unit | Quux (Optional Int64)
```

These are all valid JSON encodings for values of type Foo:

```
{"tag": "Bar", "value": 42}
{"tag": "Baz", "value": {}}
{"tag": "Quux", "value": null}
{"tag": "Quux", "value": 42}
```

Note that Daml data types with named fields are compiled by factoring out the record. So for example if we have

```
data Foo = Bar {f1: Int64, f2: Bool} | Baz
```

We'll get in Daml-LF

```
record Foo.Bar = {f1: Int64, f2: Bool}
variant Foo = Bar Foo.Bar | Baz Unit
```

and then, from JSON

```
{"tag": "Bar", "value": {"f1": 42, "f2": true}}
{"tag": "Baz", "value": {}}
```

This can be encoded and used in TypeScript, including exhaustiveness checking; see a type refinement example.

Enum

Enums are represented as strings. So if we have

```
enum Foo = Bar | Baz
```

There are exactly two valid JSON values for Foo, `Bar` and `Baz`.

1.12.4.3 Query Language

The body of `POST /v1/query` looks like so:

```
{
    "templateIds": [...template IDs...],
    "query": {...query elements...}
}
```

The elements of that query are defined here.

Fallback Rule

Unless otherwise required by one of the other rules below or to follow, values are interpreted according to *Daml-LF JSON Encoding*, and compared for equality.

All types are supported by this simple equality comparison except:

- lists
- textmaps
- genmaps

Simple Equality

Match records having at least all the (potentially nested) keys expressed in the query. The result record may contain additional properties.

Example: `{ person: { name: "Bob" }, city: "London" }`

Match: `{ person: { name: "Bob", dob: "1956-06-21" }, city: "London", createdAt: "2019-04-30T12:34:12Z" }`
No match: `{ person: { name: "Bob" }, city: "Zurich" }`

Typecheck failure: `{ person: { name: ["Bob", "Sue"] }, city: "London" }`

A JSON object, when considered with a record type, is always interpreted as a field equality query. Its type context is thus mutually exclusive with comparison queries.

Comparison Query

Match values on comparison operators for int64, numeric, text, date, and time values. Instead of a value, a key can be an object with one or more operators: `{ <op>: value }` where `<op>` can be:

"`%lt`" for less than
"`%gt`" for greater than
"`%lte`" for less than or equal to
"`%gte`" for greater than or equal to

"`%lt`" and "`%lte`" may not be used at the same time, and likewise with "`%gt`" and "`%gte`", but all other combinations are allowed.

Example: `{ "person" { "dob": { "%lt": "2000-01-01", "%gte": "1980-01-01" } } }`

Match: `{ person: { dob: "1986-06-21" } }`
No match: `{ person: { dob: "1976-06-21" } }`
No match: `{ person: { dob: "2006-06-21" } }`

These operators cannot occur in objects interpreted in a record context, nor may other keys than these four operators occur where they are legal, so there is no ambiguity with field equality.

Appendix: Type-aware Queries

This section is non-normative.

This is not a *JSON* query language, it is a *Daml-LF* query language. So, while we could theoretically treat queries (where not otherwise interpreted by the may contain additional properties rule above) without concern for what LF type (i.e. template) we're considering, we *will not* do so.

Consider the subquery `{"foo": "bar"}`. This query conforms to types, among an unbounded number of others:

```
record A □ { foo : Text }
record B □ { foo : Optional Text }
variant C □ foo : Party | bar : Unit

// NB: LF does not require any particular case for VariantCon or Field;
// these are perfectly legal types in Daml-LF packages
```

In the cases of `A` and `B`, "`foo`" is part of the query language, and only "`bar`" is treated as an LF value; in the case of `C`, the whole query is treated as an LF value. The wide variety of ambiguous interpretations about what elements are interpreted, and what elements treated as literal, and *how* those elements are interpreted or compared, would preclude many techniques for efficient query compilation and LF value representation that we might otherwise consider.

Additionally, it would be extremely easy to overlook unintended meanings of queries when writing them, and impossible in many cases to suppress those unintended meanings within the query language. For example, there is no way that the above query could be written to match `A` but never `C`.

For these reasons, as with LF value input via JSON, queries written in JSON are also always interpreted with respect to some specified LF types (e.g. template IDs). For example:

```
{
    "templateIds": ["Foo:A", "Foo:B", "Foo:C"],
    "query": {"foo": "bar"}
}
```

will treat `"foo"` as a field equality query for A and B, and (supposing templates' associated data types were permitted to be variants, which they are not, but for the sake of argument) as a whole value equality query for C.

The above Typecheck failure happens because there is no LF type to which both `"Bob"` and `["Bob", "Sue"]` conform; this would be caught when interpreting the query, before considering any contracts.

Appendix: Known Issues

When Using Oracle, Queries Fail if a Token Is Too Large

This limitation is exclusive to users of the HTTP JSON API using Daml Enterprise support for Oracle. Due to a known limitation in Oracle, the full-test JSON search index on the contract payloads rejects query tokens larger than 256 bytes. This limitations shouldn't impact most workloads, but if this needs to be worked around, the HTTP JSON API server can be started passing the additional `disableContractPayloadIndexing=true` (after wiping an existing query store database, if necessary).

Issue on GitHub

1.12.4.4 Using JavaScript Client Libraries with Daml

The JavaScript client libraries allow you to easily build frontend applications that interact with the *HTTP JSON API service*.

These libraries can dramatically reduce the time necessary to develop a full-stack application by abstracting away implementation details, particularly when building a prototype or an application with relatively simple requirements.

The @daml/types library contains the TypeScript data types corresponding to primitive Daml data types, such as `Party` or `Text`. Apart from its usefulness for TypeScript developers, the library can also be pulled in as a development-type dependency for JavaScript projects to take advantage of tooling integration with the TypeScript ecosystem, such as the availability of autocompletion on Visual Studio Code.

The @daml/ledger library contains functions used to interact with the endpoints exposed by HTTP JSON API service and forms the basic layer of functionality. At this layer, you can easily query for active contracts from the ledger, create new ones or exercise choices. This layer is agnostic with regards to any specific framework required to build the frontend.

Finally, if you are a React.js user, you can take advantage of the @daml/react library, which builds on top of `@daml/ledger` with extensions specific to React.js. This bridges the gap between the basic functionality and the infrastructure required to build a React.js-based frontend application. If you

want to start from a ready-made application that uses this library you can start running from the following template:

```
daml new --template create-daml-app <name-of-your-project>
```

To use these libraries, you need to use the *JavaScript Code Generator* to automatically generate TypeScript containing metadata about Daml packages.

Use the JavaScript Code Generator

The command `daml codegen js` generates JavaScript (and TypeScript) that can be used in conjunction with the JavaScript Client Libraries for interacting with a Daml ledger via the HTTP JSON API.

Inputs to the command are DAR files. Outputs are JavaScript packages with TypeScript typings containing metadata and types for all Daml packages included in the DAR files.

The generated packages use the library @daml/types.

Generate and Use Code

In outline, the command to generate JavaScript and TypeScript typings from Daml is `daml codegen js -o OUTDIR DAR` where `DAR` is the path to a DAR file (generated via `daml build`) and `OUTDIR` is a directory where you want the artifacts to be written.

Here's a complete example on a project built from the standard skeleton template.

```
daml new my-proj --template skeleton # Create a new project based off the
↪skeleton template
cd my-proj # Enter the newly created project directory
daml build  # Compile the project's Daml files into a DAR
daml codegen js -o daml.js .daml/dist/my-proj-0.0.1.dar # Generate JavaScript
↪packages in the daml.js directory
```

On execution of these commands:

- The directory `my-proj/daml.js` contains generated JavaScript packages with TypeScript typings;
- The files are arranged into directories;
- One of those directories will be named my-proj-0.0.1 and will contain the definitions corresponding to the Daml files in the project;
- For example, `daml.js/my-proj-0.0.1/lib/index.js` provides access to the definitions for `daml/Main.daml`;
- The remaining directories correspond to modules of the Daml standard library;
- Those directories have numeric names (the names are hashes of the Daml-LF package they are derived from).

To get a quickstart idea of how to use what has been generated, you may wish to jump to the *Templates and choices* section and return to the reference material that follows as needed.

Primitive Daml Types: @daml/types

To understand the TypeScript typings produced by the code generator, it is helpful to keep in mind this quick review of the TypeScript equivalents of the primitive Daml types provided by @daml/types.

Interfaces:

```
Template<T extends object, K = unknown>
Choice<T extends object, C, R, K = unknown>
```

Types:

Daml	TypeScript	TypeScript definition
`()`	`Unit`	`{}`
`Bool`	`Bool`	`boolean`
`Int`	`Int`	`string`
`Decimal`	`Decimal`	`string`
`Numeric ν`	`Numeric`	`string`
`Text`	`Text`	`string`
`Time`	`Time`	`string`
`Party`	`Party`	`string`
`[τ]`	`List<τ>`	`τ[]`
`Date`	`Date`	`string`
`ContractId τ`	`Contract-tId<τ>`	`string`
`Optional τ`	`Optional<τ>`	`null | (null extends τ ? [] | [Exclude<τ, null>] : τ)`
`TextMap τ`	`TextMap<τ>`	`{ [key: string]: τ }`
`(τ□, τ□)`	`Tuple□<τ□, τ□>`	`{_1: τ□; _2: τ□}`

Note: The types given in the TypeScript column are defined in @daml/types.

Note: For *n*-tuples where *n* 3, representation is analogous with the pair case (the last line of the table).

Note: The TypeScript types `Time`, `Decimal`, `Numeric` and `Int` all alias to `string`. These choices relate to the avoidance of precision loss under serialization over the json-api.

Note: The TypeScript definition of type `Optional<τ>` in the above table might look complicated. It accounts for differences in the encoding of optional values when nested versus when they are not (i.e. top-level). For example, `null` and `"foo"` are two possible values of `Optional<Text>` whereas, `[]` and `["foo"]` are two possible values of type `Optional<Optional<Text>>` (`null` is another possible value, `[null]` is **not**).

Daml to TypeScript Mappings

The mappings from Daml to TypeScript are best explained by example.

Records

In Daml, we might model a person like this.

```
data Person =
  Person with
    name: Text
    party: Party
    age: Int
```

Given the above definition, the generated TypeScript code will be as follows.

```
type Person = {
  name: string;
  party: daml.Party;
  age: daml.Int;
}
```

Variants

This is a Daml type for a language of additive expressions.

```
data Expr a =
    Lit a
  | Var Text
  | Add (Expr a, Expr a)
```

In TypeScript, it is represented as a discriminated union.

```
type Expr<a> =
  |  { tag: 'Lit'; value: a }
  |  { tag: 'Var'; value: string }
  |  { tag: 'Add'; value: {_1: Expr<a>, _2: Expr<a>} }
```

Sum-of-products

Let's slightly modify the `Expr a` type of the last section into the following.

```
data Expr a =
    Lit a
  | Var Text
  | Add {lhs: Expr a, rhs: Expr a}
```

Compared to the earlier definition, the `Add` case is now in terms of a record with fields `lhs` and `rhs`. This renders in TypeScript like so.

```
type Expr<a> =
  |  { tag: 'Lit2'; value: a }
  |  { tag: 'Var2'; value: string }
  |  { tag: 'Add'; value: Expr.Add<a> }

namespace Expr {
  type Add<a> = {
    lhs: Expr<a>;
    rhs: Expr<a>;
  }
}
```

The thing to note is how the definition of the `Add` case has given rise to a record type definition `Expr.Add`.

Enums

Given a Daml enumeration like this,

```
data Color = Red | Blue | Yellow
```

the generated TypeScript will consist of a type declaration and the definition of an associated companion object.

```
type Color = 'Red' | 'Blue' | 'Yellow'

const Color = {
  Red: 'Red',
  Blue: 'Blue',
  Yellow: 'Yellow',
  keys: ['Red','Blue','Yellow'],
} as const;
```

Templates and Choices

Here is a Daml template of a basic 'IOU' contract.

```
template Iou
  with
    issuer: Party
    owner: Party
    currency: Text
    amount: Decimal
  where
    signatory issuer
    choice Transfer: ContractId Iou
      with
        newOwner: Party
      controller owner
      do
        create this with owner = newOwner
```

The `daml codegen js` command generates types for each of the choices defined on the template as well as the template itself.

```
type Transfer = {
  newOwner: daml.Party;
}

type Iou = {
  issuer: daml.Party;
  owner: daml.Party;
  currency: string;
  amount: daml.Numeric;
}
```

Each template results in the generation of a companion object. Here, is a schematic of the one generated from the `Iou` template[2].

```
const Iou: daml.Template<Iou, undefined> & {
  Archive: daml.Choice<Iou, DA_Internal_Template.Archive, {}, undefined>;
  Transfer: daml.Choice<Iou, Transfer, daml.ContractId<Iou>, undefined>;
} = {
  /* ... */
}
```

The exact details of these companion objects are not important - think of them as representing metadata .

What **is** important is the use of the companion objects when creating contracts and exercising choices using the @daml/ledger package. The following code snippet demonstrates their usage.

```
1  import Ledger from  '@daml/ledger';
2  import {Iou, Transfer} from /* ... */;
3
4  const ledger = new Ledger(/* ... */);
5
6  // Contract creation; Bank issues Alice a USD $1MM IOU.
7
8  const iouDetails: Iou = {
9    issuer: 'Chase',
10   owner: 'Alice',
11   currency: 'USD',
12   amount: 1000000.0,
13 };
14 const aliceIouCreateEvent = await ledger.create(Iou, iouDetails);
15 const aliceIouContractId = aliceIouCreateEvent.contractId;
16
17 // Choice execution; Alice transfers ownership of the IOU to Bob.
18
19 const transferDetails: Transfer = {
20   newOwner: 'Bob',
21 }
22 const [bobIouContractId, _] = await ledger.exercise(Transfer, aliceIouContractId,
   ↪transferDetails);
```

Observe on line 14, the first argument to `create` is the `Iou` companion object and on line 22, the first argument to `exercise` is the `Transfer` companion object.

[2] The `undefined` type parameter captures the fact that `Iou` has no contract key.

@daml/react

@daml/react documentation

@daml/ledger

@daml/ledger documentation

@daml/types

@daml/types documentation

1.12.4.5 JSON API Production Setup

Production Setup

The vast majority of prior documentation focused on ease of testing and on setting up the service to run in a dev environment. From a production perspective, given the wide variety of use-cases, there is far less of an established framework for the deployment of an *HTTP JSON API* server. In this document we will make some recommendations for production deployments.

Query Store

Note: Daml Open Source only supports PostgreSQL backends for the *HTTP JSON API* server, but Daml Enterprise also supports Oracle backends.

The *HTTP JSON API* server is a JVM application that uses an in-memory backend by default. This in-memory backend setup is inefficient for larger datasets as every query fetches the entire active contract set for all the templates the query references. For production setups we therefore recommend, at a minimum, that one use a database as a query store. This allows for more efficient data caching and improves query performance. Details for enabling a query store are given below.

The query store is a cached search index and is useful in cases where the application needs to query large active contract sets (ACS). The *HTTP JSON API* server can be configured with PostgreSQL/Oracle (Daml Enterprise only) as the query store backend.

The query store is built by saving the state of the ACS up to the current ledger offset. This allows the *HTTP JSON API* to only request the delta on subsequent queries, making it much faster than requesting the entire ACS every time.

Configuring

For example, to enable the PostgreSQL backend you can add the `query-store` config block in your application config file:

```
query-store {
  base-config {
    user = "postgres"
    password = "password"
    driver = "org.postgresql.Driver"
    url = "jdbc:postgresql://localhost:5432/test?&ssl=true"

    // prefix for table names to avoid collisions, empty by default
    table-prefix = "foo"

    // max pool size for the database connection pool
    pool-size = 12
    //specifies the min idle connections for database connection pool.
    min-idle = 4
    //specifies the idle timeout for the database connection pool.
    idle-timeout = 12s
    //specifies the connection timeout for database connection pool.
    connection-timeout = 90s
  }
  // option setting how the schema should be handled.
  // Valid options are start-only, create-only, create-if-needed-and-start and
↪create-and-start
  start-mode = "start-only"
}
```

Consult your database vendor's JDBC driver documentation to learn how to specify a JDBC connection URL that suits your needs.

You can also use the `--query-store-jdbc-config` CLI flag (deprecated), as shown below.

```
daml json-api --ledger-host localhost --ledger-port 6865 --http-port 7575 \
--query-store-jdbc-config "driver=org.postgresql.Driver,url=jdbc:postgresql://
↪localhost:5432/test?&ssl=true,user=postgres,password=password,start-mode=start-
↪only"
```

Managing DB permissions with `start-mode`

The `start-mode` is a custom parameter to specify the initialization and usage of the database backing the query store.

Depending on how you prefer to operate it, you can

run with `start-mode=create-only` with a user that has exclusive table-creating rights that are required for the query store to operate, and then start it once more with `start-mode=start-only` with a user that can use the aforementioned tables, but that cannot apply schema changes
run with a user that can both create and use the query store tables by passing `start-mode=create-and-start`
run with a user that can drop, create and use the query store tables by passing `start-mode=create-if-needed-and-start`

When restarting the *HTTP JSON API* server after a schema has already been created, it's safe practice to always use `start-mode=start-only`.

Data Continuity

The query store is a cache. This means that it is perfectly fine to drop it, as the data it contains is a subset of what can safely be recovered from the ledger.

As such, the query store does not provide data continuity guarantees across versions and furthermore doesn't guarantee that a query store initialized with a previous version of the *HTTP JSON API* will work with a newer version. However, the query store keeps track of the schema version under which it was initialized and HTTP JSON API service refuses to start if an old schema is detected when it's run with a newer version.

To evolve, the operator of the *HTTP JSON API* query store needs to drop the database used to hold the *HTTP JSON API* query store, create a new one (consult your database vendor's documentation for instructions), and then (depending on the operator's preferred production setup) should proceed to create and start the server using either `start-mode=create-only` & `start-mode=start-only` or only with `start-mode=create-and-start` as described above.

Behavior Under High Load

As stated *in the overview*, the HTTP JSON API service is optimized for rapid application development and ease of developer onboarding. It is not intended to support every high-performance use case. To understand how a high-load application may reach the limits of its design, you need to consider how the query store works.

First, always keep in mind that *the HTTP JSON API service can only do whatever an ordinary ledger API client application could do, including your own.* That's because it *is* an ordinary client of *The Ledger API*. So, if your application's queries are a poor match for the way HTTP JSON API service's query store works, it's time to consider cutting out the middleman.

Running a Query

Here is what happens every time you run a query with a configured query store:

1. The query store uses the transaction stream from the gRPC API to update its contract table with an up-to-date view of all active contracts that match the template IDs, interface IDs, and user party set in the request. The payload query is not considered at all; every matching contract is added to the table. This will use the active contract service to skip past most of the transaction stream, if the contract table is empty at that set.
2. A database query is run on the contract table, filtering on template ID/interface ID, party set, and the payload.
3. If contention with concurrent requests is detected, the query store will assume it is behind and catch up by returning to #1. This uses an iterative livelocking strategy, where progress is guaranteed and more concurrency is permitted, rather than exclusive locking.
4. Results are returned to the user.

A websocket query does the same, but any contract that didn't exist at the *start* of the websocket won't receive the above treatment; the live data described for the websocket query stream is always filtered directly from the gRPC API, just as if no query store was configured.

Storage Overview

Without going into too much detail, here's more or less what is stored under step #1 above, *for each contract*:

1. full contract ID
2. an integer for the template or interface ID
3. for a template ID, the create arguments, as full JSON
4. for an interface ID, the interface view, as full JSON
5. a list of signatories and observers, i.e. parties

Every query store backend indexes on #2, as we have found this index to be universally beneficial. In addition, the Oracle backend has an index on #3 and #4.

With this indexing arrangement, our testing has indicated reasonable performance for well-matched use cases as explained below for contract tables of up to 100000 contracts.

Well-Matched Use Cases

The query store is, generally speaking, best matched to CRUD-like use cases with relatively stable active contract sets. Here are some more specific characteristics likely to be shared by Daml designs that will perform well with the query store.

1. Workflows properly separated into separate templates. The template ID index is the most efficient part of query store filtering. In addition, contract table updates on separate template IDs do not contend (i.e. cause the reset to step #1 above), so changes to the ledger on other parts of the workflow do not affect queries on the template in question.
2. Queries that return <10% of all active contracts for a given contract type ID and party set. This maximizes the value of storing redundant copies in SQL-queryable form at all, namely, that the HTTP JSON API service does not even need to consider already-stored, unmatched contracts.
3. Queries against a slow participant. If the transaction stream from your ledger API participant server is particularly slow, it may be faster to retrieve most contracts from its local database, even if HTTP JSON API service gets no benefit from #2.
4. Templates with low churn, i.e. most active contracts from the previous query are likely to still be active for the next query. If the query store is likelier to have already stored most of the contracts for that template, the update part of the process will be significantly faster and much less likely to contend.

Ill-Matched Use Cases

By contrast, many Daml applications can yield patterns in the ACS and transactions that hurt the performance of applications built on the HTTP JSON API service. Below are some gotchas that might indicate that your application calls for a custom view, perhaps even stored locally in SQL and managed by your application, beyond what HTTP JSON API service's query store can provide.

1. Workflows that use the state field antipattern. This adds a filter on the relatively inefficient payload query that ought to instead be placed on the template ID. In addition, updates to the state field will needlessly contend with updates to contracts with the state you're interested in.
2. Queries that return a large percentage of active contracts against a given contract type ID and party set. If the query store cannot yield any benefit from letting HTTP JSON API service ignore most contracts on each query it will spend more time updating its contract table than it would

have spent simply reading from the gRPC API and filtering directly, so you might as well turn off the query store.
3. Templates with high churn, i.e. the active contracts during the last query are very unlikely to still be active. In such cases HTTP JSON API service may spend so much time updating its contract table that it washes out any performance advantage from being able to SQL query it afterwards.
4. Contracts with highly-overlapping signatories and observers. When signatories and observers do not intersect, their updates never contend; the more this happens, the more likely updates for queries with different party-sets will contend.

Security and Privacy

For an *HTTP JSON API* server, all data is maintained by the operator of the deployment. It is the operator's responsibility to ensure that the data abides by the necessary regulations and confidentiality expectations.

We recommend using the tools documented by PostgreSQL to protect data at rest, and using a secure communication channel between the *HTTP JSON API* server and the PostgreSQL server.

The *HTTP JSON API* server provides TLS support to protect data in transit and over untrusted networks. To enable TLS you must specify both the private key for your server and the certificate chain via the below config block that specifies the `cert-chain-file`, `private-key-file`. You can also set a custom root CA certificate that will be used to validate client certificates via the `trust-collection-file` parameter:

```
ledger-api {
  address = "127.0.0.1"
  port = 6400
  tls {
    enabled = "true"
    // the certificate to be used by the server
    cert-chain-file = "cert-chain.crt"
    // private key of the server
    private-key-file = "pvt-key.pem"
    // trust collection, which means that all client certificates will be
↪verified using the trusted
    // certificates in this store. if omitted, the JVM default trust store is
↪used.
    trust-collection-file = "root-ca.crt"
  }
}
```

Using the cli options (deprecated), you can specify tls options using``daml json-api -pem server.pem -crt server.crt``. Custom root CA certificate can be set via `--cacrt ca.crt`

For more details on secure Daml infrastructure setup please see this reference implementation

Architecture

Components

A production setup of the *HTTP JSON API* involves the following components:

the *HTTP JSON API* server
the query store backend database server
the ledger

The *HTTP JSON API* server exposes an API to interact with the Ledger. It uses JDBC to interact with its underlying query store in order to cache and serve data efficiently.

The *HTTP JSON API* server releases are regularly tested with the tools described under *System Requirements*.

In production, we recommend running on a x86_64 architecture in a Linux environment. This environment should have a Java SE Runtime Environment with minimum version as mentioned at *System Requirements*. We recommend using PostgreSQL server as query-store, again with minimum version as mentioned at *System Requirements*.

Scaling and Redundancy

Note: This section of the document only talks about scaling and redundancy setup for the *HTTP JSON API* server. In all recommendations suggested below we assume that the JSON API is always interacting with a single participant on the ledger.

We recommend dedicating computation and memory resources to the *HTTP JSON API* server and query store components. This can be achieved via containerization or by setting these components up on independent physical servers. Make sure that the two components are **physically co-located** to reduce network latency for communication. Scaling and availability heavily rely on the interactions between the core components listed above.

The general principles of scaling apply here: Try to understand the bottlenecks and see if adding additional processing power/memory helps.

Scaling creates and exercises

The HTTP JSON API service provides simple, synchronous endpoints for carrying out creates and exercises on the ledger. It does not support the complex multi-command asynchronous submission protocols supported by the ledger API.

For performing large numbers of creates and exercises at once, while you can perform many HTTP requests at once to carry out this task, it may be simpler and more concurrent-safe to shift more of this logic into a Daml choice that can be exercised.

The pattern looks like this:

1. Have a contract with a key and one or more choices on the ledger.
2. Such a choice can carry out as many creates and exercises as desired; all of these will take place in a single transaction.

3. Use the HTTP JSON API service to exercise this choice by key.

It's possible to go too far in the other direction: any error will usually cause the whole transaction to roll back, so an excessively large amount of work done by a single choice can also cause needless retrying. You can solve this by batching requests, or using *Exception Handling* to collect and return failed cases to the HTTP JSON API service client for retrying, allowing successful parts of the batch to proceed.

Scaling Queries

The *Query Store* is a key factor of efficient queries. However, it behaves very differently depending on the characteristics of the underlying ledger, Daml application, and client query patterns. *Understanding how it works* is a major prerequisite to understanding how the HTTP JSON API service will interact with your application's performance profile.

Additionally, the *HTTP JSON API* can be scaled independently of its query store. You can have any number of *HTTP JSON API* instances talking to the same query store (if, for example, your monitoring indicates that the *HTTP JSON API* processing time is the bottleneck), or have each HTTP JSON API instance talk to its own independent query store (if the database response times are the bottleneck).

In the latter case, the Daml privacy model ensures that the *HTTP JSON API* requests are made using the user-provided token, thus the data stored in a given query store will be specific to the set of parties that have made queries through that specific query store instance (for a given template). Therefore, if you do run with separate query stores, it may be useful to route queries (using a reverse proxy server) based on requesting party (and possibly queried template), which would minimize the amount of data in each query store as well as the overall redundancy of said data.

Users may consider running PostgreSQL backend in a high availability configuration. The benefits of this are use-case dependent as this may be more expensive for smaller active contract datasets, where re-initializing the cache is cheap and fast.

Finally, we recommend using orchestration systems or load balancers which monitor the health of the service and perform subsequent operations to ensure availability. These systems can use the healthcheck endpoints provided by the *HTTP JSON API* server. This can also be tied into supporting an arbitrary autoscaling implementation in order to ensure a minimum number of *HTTP JSON API* servers on failures.

Hitting a Scaling Bottleneck

As HTTP JSON API service and its query store are optimized for rapid application development and ease of developer onboarding, you may reach a point where your application's performance demands exceed what the HTTP JSON API service can offer. The more demanding your application is, the less likely it is to be well-matched with the simplifications and generalizations that the HTTP JSON API service makes for developer simplicity.

In this case, it's important to remember that *the HTTP JSON API service can only do whatever an ordinary ledger API client application could do, including your own.*

For example, for a JVM application, interacting with JSON is probably simpler than gRPC directly, but using *Java Bindings codegen* are much simpler than either.

There is no way to make *Query Store* more suited to high-performance queries for your Daml application than a custom data store implemented as your own server on gRPC would be. So an application

that *must* interact over JSON, but requires very high-performance or very high-load query throughput, would usually be better served by a custom server.

Set Up the HTTP JSON API Service To Work With Highly Available Participants

If the participant node itself is configured to be highly available, depending on the setup you may want to choose different approaches to connect to the passive participant node(s). In most setups, including those based on Canton, you'll likely have an active participant node whose role can be taken over by a passive node in case the currently active one drops. Just as for the *HTTP JSON API* itself, you can use orchestration systems or load balancers to monitor the status of the participant nodes and have those point your (possibly highly-available) *HTTP JSON API* nodes to the active participant node.

To learn how to run and monitor Canton with high availability, refer to the *Canton documentation*.

Logging

The *HTTP JSON API* server uses the industry-standard logback for logging. You can read more about it in the Logback documentation.

The logging infrastructure leverages structured logging as implemented by the Logstash Logback Encoder.

Logged events should carry information about the request being served by the *HTTP JSON API* server. This includes the details of the commands being submitted, the endpoints being hit, and the response received – highlighting details of failures if any. When using a traditional logging target (e.g. standard output or rotating files) this information will be part of the log description. Using a logging target compatible with the Logstash Logback Encoder allows one to have rich logs that come with structured information about the event being logged.

The default log encoder used is the plaintext one for traditional logging targets.

Metrics

Enable and Configure Reporting

To enable metrics and configure reporting, you can use the below config block in application config:

```
metrics {
  // Start a metrics reporter. Must be one of "console", "csv:///PATH",
↪"graphite://HOST[:PORT][/METRIC_PREFIX]", or "prometheus://HOST[:PORT]".
  reporter = "prometheus://localhost:9000"
  // Set metric reporting interval , examples : 1s, 30s, 1m, 1h
  reporting-interval = 30s
}
```

or the two following CLI options (deprecated):

`--metrics-reporter`: passing a legal value will enable reporting; the accepted values are as follows:

- `console`: prints captured metrics on the standard output
- `csv://</path/to/metrics.csv>`: saves the captured metrics in CSV format at the specified location

- `graphite://<server_host>[:<server_port>]`: sends captured metrics to a Graphite server. If the port is omitted, the default value `2003` will be used.
- `prometheus://<server_host>[:<server_port>]`: renders captured metrics on a HTTP endpoint in accordance with the Prometheus protocol. If the port is omitted, the default value `55001` will be used. The metrics will be available under the address `http://<server_host>:<server_port>/metrics`.

`--metrics-reporting-interval`: allows the user to set the interval at which metrics are pre-aggregated on the *HTTP JSON API* and sent to the reporter. The formats accepted are based on the ISO 8601 duration format `PnDTnHnMn.nS` with days considered to be exactly 24 hours. The default interval is 10 seconds.

Types of Metrics

This is a list of type of metrics with all data points recorded for each. Use this as a reference when reading the list of metrics.

Counter

Number of occurrences of some event.

Meter

A meter tracks the number of times a given event occurred (throughput). The following data points are kept and reported by any meter.

`<metric.qualified.name>.count`: number of registered data points overall
`<metric.qualified.name>.m1_rate`: number of registered data points per minute
`<metric.qualified.name>.m5_rate`: number of registered data points every 5 minutes
`<metric.qualified.name>.m15_rate`: number of registered data points every 15 minutes
`<metric.qualified.name>.mean_rate`: mean number of registered data points

Timers

A timer records the time necessary to execute a given operation (in fractional milliseconds).

Metrics Reference

The HTTP JSON API Service supports *common HTTP metrics*. In addition, see the following list of important metrics:

`daml.http_json_api.incoming_json_parsing_and_validation_timing`

A timer. Measures latency (in milliseconds) for parsing and decoding of an incoming json payload

`daml.http_json_api.response_creation_timing`

A timer. Measures latency (in milliseconds) for construction of the response json payload.

`daml.http_json_api.db_find_by_contract_key_timing`

A timer. Measures latency (in milliseconds) of the find by contract key database operation.

`daml.http_json_api.db_find_by_contract_id_timing`

A timer. Measures latency (in milliseconds) of the find by contract id database operation.

`daml.http_json_api.command_submission_ledger_timing`

A timer. Measures latency (in milliseconds) for processing the command submission requests on the ledger.

`daml.http_json_api.websocket_request_count`

A Counter. Counts active websocket connections.

1.12.5 The Ledger API

To write an application around a Daml ledger, you will need to interact with the **Ledger API**.

Every ledger that Daml can run on exposes this same API.

1.12.5.1 What's in the Ledger API

The Ledger API exposes the following services:

Submitting commands to the ledger
- Use the *command submission service* to submit commands (create a contract or exercise a choice) to the ledger.
- Use the *command completion service* to track the status of submitted commands.
- Use the *command service* for a convenient service that combines the command submission and completion services.

Reading from the ledger
- Use the *transaction service* to stream committed transactions and the resulting events (choices exercised, and contracts created or archived), and to look up transactions.

- Use the *active contracts service* to quickly bootstrap an application with the currently active contracts. It saves you the work to process the ledger from the beginning to obtain its current state.

Utility services

- Use the *party management service* to allocate and find information about parties on the Daml ledger.
- Use the *package service* to query the Daml packages deployed to the ledger.
- Use the *ledger identity service* to retrieve the Ledger ID of the ledger the application is connected to.
- Use the *ledger configuration service* to retrieve some dynamic properties of the ledger, like maximum deduplication duration for commands.
- Use the *version service* to retrieve information about the Ledger API version.
- Use the *user management service* to manage users and their rights.
- Use the *metering report service* to retrieve a participant metering report.

Testing services (on Sandbox only, *not* for production ledgers)

- Use the *time service* to obtain the time as known by the ledger.

For full information on the services see *The Ledger API Services*.

You may also want to read the *protobuf documentation*, which explains how each service is defined as protobuf messages.

1.12.5.2 How to Access the Ledger API

You can access the Ledger API via the *Java Bindings* or the *Python Bindings* (formerly known as DAZL).

If you don't use a language that targets the JVM or Python, you can use gRPC to generate the code to access the Ledger API in several supported programming languages. *Further documentation* provides a few pointers on how you may want to approach this.

You can also use the *HTTP JSON API Service* to tap into the Ledger API.

At its core, this service provides a simplified view of the active contract set and additional primitives to query it and exposing it using a well-defined JSON-based encoding over a conventional HTTP connection.

A subset of the services mentioned above is also available as part of the HTTP JSON API.

1.12.5.3 Daml-LF

When you *compile Daml source into a .dar file*, the underlying format is Daml-LF. Daml-LF is similar to Daml, but is stripped down to a core set of features. The relationship between the surface Daml syntax and Daml-LF is loosely similar to that between Java and JVM bytecode.

As a user, you don't need to interact with Daml-LF directly. But internally, it's used for:

Executing Daml code on the Sandbox or on another platform
Sending and receiving values via the Ledger API (using a protocol such as gRPC)
Generating code in other languages for interacting with Daml models (often called codegen)

When You Need to Know About Daml-LF

Daml-LF is only really relevant when you're dealing with the objects you send to or receive from the ledger. If you use any of the provided language bindings for the Ledger API, you don't need to know about Daml-LF at all, because this generates idiomatic representations of Daml for you.

Otherwise, it can be helpful to know what the types in your Daml code look like at the Daml-LF level, so you know what to expect from the Ledger API.

For example, if you are writing an application that creates some Daml contracts, you need to construct values to pass as parameters to the contract. These values are determined by the Daml-LF types in that contract template. This means you need an idea of how the Daml-LF types correspond to the types in the original Daml model.

For the most part the translation of types from Daml to Daml-LF should not be surprising. *This page goes through all the cases in detail*.

For the bindings to your specific programming language, you should refer to the language-specific documentation.

1.12.5.4 The Ledger API Services

The Ledger API is structured as a set of services. The core services are implemented using gRPC and Protobuf, but most applications access this API through the mediation of the language bindings.

This page gives more detail about each of the services in the API, and will be relevant whichever way you're accessing it.

If you want to read low-level detail about each service, see the *protobuf documentation of the API*.

Overview

The API is structured as two separate data streams:

> A stream of **commands** TO the ledger that allow an application to submit transactions and change state.
> A stream of **transactions** and corresponding **events** FROM the ledger that indicate all state changes that have taken place on the ledger.

Commands are the only way an application can cause the state of the ledger to change, and events are the only mechanism to read those changes.

For an application, the most important consequence of these architectural decisions and implementation is that the Ledger API is asynchronous. This means:

> The outcome of commands is only known some time after they are submitted.
> The application must deal with successful and erroneous command completions separately from command submission.
> Ledger state changes are indicated by events received asynchronously from the command submissions that cause them.

The need to handle these issues is a major determinant of application architecture. Understanding the consequences of the API characteristics is important for a successful application design.

For more help understanding these issues so you can build correct, performant and maintainable applications, read the *application architecture guide*.

Glossary

The ledger is a list of `transactions`. The transaction service returns these.
A `transaction` is a tree of `actions`, also called `events`, which are of type `create`, `exercise` or `archive`. The transaction service can return the whole tree, or a flattened list.
A `submission` is a proposed transaction, consisting of a list of `commands`, which correspond to the top-level `actions` in that transaction.
A `completion` indicates the success or failure of a `submission`.

Submit Commands to the Ledger

Command Submission Service

Use the **command submission service** to submit commands to the ledger. Commands either create a new contract, or exercise a choice on an existing contract.

A call to the command submission service will return as soon as the ledger server has parsed the command, and has either accepted or rejected it. This does not mean the command has been executed, only that the server has looked at the command and decided that its format is acceptable, or has rejected it for syntactic or content reasons.

The on-ledger effect of the command execution will be reported via the *transaction service*, described below. The completion status of the command is reported via the *command completion service*. Your application should receive completions, correlate them with command submission, and handle errors and failed commands. Alternatively, you can use the *command service*, which conveniently wraps the command submission and completion services.

Change ID

Each intended ledger change is identified by its **change ID**, consisting of the following three components:

The submitting parties, i.e., the union of *party* and *act_as*
the *application ID*
The *command ID*

Application-specific IDs

The following application-specific IDs, all of which are included in completion events, can be set in commands:

A *submission ID*, returned to the submitting application only. It may be used to correlate specific submissions to specific completions.
A *command ID*, returned to the submitting application only; it can be used to correlate commands to completions.

A *workflow ID*, returned as part of the resulting transaction to all applications receiving it. It can be used to track workflows between parties, consisting of several transactions.

For full details, see *the proto documentation for the service*.

Command Deduplication

The command submission service deduplicates submitted commands based on their *change ID*.

Applications can provide a deduplication period for each command. If this parameter is not set, the default maximum deduplication duration is used.
A command submission is considered a duplicate submission if the Ledger API server is aware of another command within the deduplication period and with the same *change ID*.
A command resubmission will generate a rejection until the original submission was rejected (i.e. the command failed and resulted in a rejected transaction) or until the effective deduplication period has elapsed since the completion of the original command, whichever comes first.
Command deduplication is only *guaranteed* to work if all commands are submitted to the same participant. Ledgers are free to perform additional command deduplication across participants. Consult the respective ledger's manual for more details.

For details on how to use command deduplication, see the *Command Deduplication Guide*.

Explicit contract disclosure (experimental)

Starting with Canton 2.7, Ledger API clients can use explicit contract disclosure to submit commands with attached disclosed contracts received from third parties. For more details, see *Explicit contract disclosure*.

Command Completion Service

Use the **command completion service** to find out the completion status of commands you have submitted.

Completions contain the *command ID* of the completed command, and the completion status of the command. This status indicates failure or success, and your application should use it to update what it knows about commands in flight, and implement any application-specific error recovery.

For full details, see *the proto documentation for the service*.

Command Service

Use the **command service** when you want to submit a command and wait for it to be executed. This service is similar to the command submission service, but also receives completions and waits until it knows whether or not the submitted command has completed. It returns the completion status of the command execution.

You can use either the command or command submission services to submit commands to effect a ledger change. The command service is useful for simple applications, as it handles a basic form

of coordination between command submission and completion, correlating submissions with completions, and returning a success or failure status. This allow simple applications to be completely stateless, and alleviates the need for them to track command submissions.

For full details, see *the proto documentation for the service*.

Read From the Ledger

Transaction Service

Use the **transaction service** to listen to changes in the ledger state, reported via a stream of transactions.

Transactions detail the changes on the ledger, and contains all the events (create, exercise, archive of contracts) that had an effect in that transaction.

Transactions contain a *transaction ID* (assigned by the server), the *workflow ID*, the *command ID*, and the events in the transaction.

Subscribe to the transaction service to read events from an arbitrary point on the ledger. This arbitrary point is specified by the ledger offset. This is important when starting or restarting and application, and to work in conjunction with the *active contracts service*.

For full details, see *the proto documentation for the service*.

Transaction and transaction Trees

`TransactionService` offers several different subscriptions. The most commonly used is `GetTransactions`. If you need more details, you can use `GetTransactionTrees` instead, which returns transactions as flattened trees, represented as a map of event IDs to events and a list of root event IDs.

Verbosity

The service works in a non-verbose mode by default, which means that some identifiers are omitted:

Record IDs
Record field labels
Variant IDs

You can get these included in requests related to Transactions by setting the `verbose` field in message `GetTransactionsRequest` or `GetActiveContractsRequest` to `true`.

Transaction Filter

`TransactionService` offers transaction subscriptions filtered by templates and interfaces using `GetTransactions` calls. A *transaction filter* in `GetTransactionsRequest`. allows:

filtering by a party, when the *inclusive* field is left empty
filtering by a party and a *template ID*
filtering by a party and an *interface ID*
exposing an interface view, when the *include_interface_view* is set to `true`

Active Contracts Service

Use the **active contracts service** to obtain a party-specific view of all contracts that are active on the ledger at the time of the request.

The active contracts service returns its response as a stream of batches of the created events that would re-create the state being reported (the size of these batches is left to the ledger implementation). As part of the last message, the offset at which the reported active contract set was valid is included. This offset can be used to subscribe to the flat transactions stream to keep a consistent view of the active contract set without querying the active contract service further.

This is most important at application start, if the application needs to synchronize its initial state with a known view of the ledger. Without this service, the only way to do this would be to read the Transaction Stream from the beginning of the ledger, which can be prohibitively expensive with a large ledger.

For full details, see *the proto documentation for the service*.

Verbosity

See *Verbosity* above.

Transaction Filter

See *Transaction Filter* above.

Note: The RPCs exposed as part of the transaction and active contracts services make use of offsets.

An offset is an opaque string of bytes assigned by the participant to each transaction as they are received from the ledger. Two offsets returned by the same participant are guaranteed to be lexicographically ordered: while interacting with a single participant, the offset of two transactions can be compared to tell which was committed earlier. The state of a ledger (i.e. the set of active contracts) as exposed by the Ledger API is valid at a specific offset, which is why the last message your application receives when calling the `ActiveContractsService` is precisely that offset. In this way, the client can keep track of the relevant state without needing to invoke the `ActiveContractsService` again, by starting to read transactions from the given offset.

Offsets are also useful to perform crash recovery and failover as documented more in depth in the *application architecture* page.

You can read more about offsets in the protobuf documentation of the API.

Event Query Service (EXPERIMENTAL)

Use the **event query service** to obtain a party-specific view of contract events.

Contract events can be queried by contract id or contract key. If the events being queried are not visible to the requesting parties, the service returns an empty structure. This service returns consumed contracts up until they are pruned.

In the case of contract keys, a number of contracts may have used the contract key over time. The latest contract is returned first, with earlier contracts being returned in subsequent calls with a populated continuation token.

Note: When querying by contract key, the key value must be structured in the same way as the key returned in the create event.

For full details, see *the proto documentation for the service*.

Utility Services

Party Management Service

Use the **party management service** to allocate parties on the ledger, update party properties local to the participant and retrieve information about allocated parties.

Parties govern on-ledger access control as per *Daml's privacy model* and *authorization rules*. Applications and their operators are expected to allocate and use parties to manage on-ledger access control as per their business requirements.

For more information, refer to the pages on *Identity Management* and *the API reference documentation*.

User Management Service

Use the **user management service** to manage the set of users on a participant node and their *access rights* to that node's Ledger API services and as the integration point for your organization's IAM (Identity and Access Management) framework.

Daml 2.0 introduced the concept of the user in Daml. While a party represents a single individual with a single set of rights and is universal across participant nodes, a user is local to a specific participant node. Each user is typically associated with a primary party and is given the right to act as or read as other parties. Every participant node will maintain its own mapping from its user ids to the parties that they can act and/or read as. Also, when used, the user's ids will serve as application ids. Thus, participant users can be used to manage the permissions of Daml applications (i.e. to authorize applications to read as or act as certain parties). Unlike a JWT token-based system, the user management system does not limit the number of parties that the user can act or read as.

The relation between a participant node's users and Daml parties is best understood by analogy to classical databases: a participant node's users are analogous to database users while Daml parties

are analogous to database roles. Further, the rights granted to a user are analogous to the user's assigned database roles.

For more information, consult the *the API reference documentation* for how to list, create, update, and delete users and their rights. See the *UserManagementFeature descriptor* to learn about the limits of the user management service, e.g., the maximum number of rights per user. The feature descriptor can be retrieved using the *Version service*.

With user management enabled you can use both new user-based and old custom Daml authorization tokens. Consult the *Authorization documentation* to understand how Ledger API requests are authorized, and how to use user management to dynamically change an application's rights.

User management is available in Canton-enabled drivers and not yet available in the Daml for VMware Blockchain driver.

Identity Provider Config Service

Use **identity provider config service** to define and manage the parameters of an external IDP systems configured to issue tokens for a participant node.

The **identity provider config service** makes it possible for participant node administrators to set up and manage additional identity providers at runtime. This allows using access tokens from identity providers unknown at deployment time. When an identity provider is configured, independent IDP administrators can manage their own set of parties and users.

Such parties and users have a matching identity_provider_id defined and are inaccessible to administrators from other identity providers. A user will only be authenticated if the corresponding JWT token is issued by the appropriate identity provider. Users and parties without identity_provider_id defined are assumed to be using the default identity provider, which is configured statically when the participant node is deployed.

For full details, see *the proto documentation for the service*.

Package Service

Use the **package service** to obtain information about Daml packages available on the ledger.

This is useful for obtaining type and metadata information that allow you to interpret event data in a more useful way.

For full details, see *the proto documentation for the service*.

Ledger Identity Service (DEPRECATED)

Use the **ledger identity service** to get the identity string of the ledger that your application is connected to.

Including identity string is optional for all Ledger API requests. If you include it, commands with an incorrect identity string will be rejected.

For full details, see *the proto documentation for the service*.

Ledger Configuration Service

Use the **ledger configuration service** to subscribe to changes in ledger configuration.

This configuration includes the maximum command deduplication period (see *Command Deduplication* for details).

For full details, see *the proto documentation for the service*.

Version Service

Use the **version service** to retrieve information about the Ledger API version and what optional features are supported by the ledger server.

For full details, see *the proto documentation for the service*.

Pruning Service

Use the **pruning service** to prune archived contracts and transactions before or at a given offset.

For full details, see *the proto documentation for the service*.

Metering Report Service

Use the **metering report service** to retrieve a participant metering report.

For full details, see *the proto documentation for the service*.

Testing Services

These are only for use for testing with the Sandbox, not for on production ledgers.

Time Service

Use the **time service** to obtain the time as known by the ledger server.

For full details, see *the proto documentation for the service*.

1.12.5.5 Java Bindings

The Java bindings is a client implementation of the *Ledger API* based on RxJava, a library for composing asynchronous and event-based programs using observable sequences for the Java VM. It provides an idiomatic way to write Daml Ledger applications.

See also:

This documentation for the Java bindings API includes the JavaDoc reference documentation.

Overview

The Java bindings library is composed of:

The Data Layer A Java-idiomatic layer based on the Ledger API generated classes. This layer simplifies the code required to work with the Ledger API.
Can be found in the java package `com.daml.ledger.javaapi.data`.

The Reactive Layer A thin layer built on top of the Ledger API services generated classes.
For each Ledger API service, there is a reactive counterpart with a matching name. For instance, the reactive counterpart of `ActiveContractsServiceGrpc` is `ActiveContractsClient`.
The Reactive Layer also exposes the main interface representing a client connecting via the Ledger API. This interface is called `LedgerClient` and the main implementation working against a Daml Ledger is the `DamlLedgerClient`.
Can be found in the java package `com.daml.ledger.rxjava`.

Generate Code

When writing applications for the ledger in Java, you want to work with a representation of Daml templates and data types in Java that closely resemble the original Daml code while still being as true to the native types in Java as possible.

To achieve this, you can use Daml to Java code generator (Java codegen) to generate Java types based on a Daml model. You can then use these types in your Java code when reading information from and sending data to the ledger.

For more information on Java code generation, see *Generate Java Code from Daml*.

Connect to the Ledger: `LedgerClient`

Connections to the ledger are made by creating instance of classes that implement the interface `LedgerClient`. The class `DamlLedgerClient` implements this interface, and is used to connect to a Daml ledger.

This class provides access to the ledgerId, and all clients that give access to the various ledger services, such as the active contract set, the transaction service, the time service, etc. This is described *below*. Consult the JavaDoc for DamlLedgerClient for full details.

Reference Documentation

Click here for the JavaDoc reference documentation.

Get Started

The Java bindings library can be added to a Maven project.

Set Up a Maven Project

To use the Java bindings library, add the following dependencies to your project's `pom.xml`:

```
<dependencies>
    <dependency>
        <groupId>com.daml</groupId>
        <artifactId>bindings-rxjava</artifactId>
        <version>x.y.z</version>
    </dependency>
</dependencies>
```

Replace `x.y.z` for both dependencies with the version that you want to use. You can find the available versions by checking the Maven Central Repository.

You can also take a look at the `pom.xml` file from the *quickstart project*.

Connect to the Ledger

Before any ledger services can be accessed, you must establish a connection to the ledger by creating an instance of a `DamlLedgerClient`. To create an instance of a ledger client, use the static `newBuilder(..)` method to create a `DamlLedgerClient.Builder`. Then use the builder instance to create the `DamlLedgerClient`. Finally, call the `connect()` method on the client.

```
// Create a client object to access services on the ledger.
DamlLedgerClient client = DamlLedgerClient.newBuilder(ledgerhost, ledgerport).
↪build();

// Connects to the ledger and runs initial validation.
client.connect();
```

Perform Authorization

Some ledgers will require you to send an access token along with each request.

To learn more about authorization, read the *Authorization* overview.

To use the same token for all Ledger API requests, the `DamlLedgerClient` builders expose a `withAccessToken` method. This will allow you to not pass a token explicitly for every call.

If your application is long-lived and your tokens are bound to expire, you can reload the necessary token when needed and pass it explicitly for every call. Every client method has an overload that allows a token to be passed, as in the following example:

```
transactionClient.getLedgerEnd(); // Uses the token specified when constructing
↪the client
transactionClient.getLedgerEnd(accessToken); // Override the token for this call
↪exclusively
```

If you're communicating with a ledger that verifies authorization it's very important to secure the communication channel to prevent your tokens to be exposed to man-in-the-middle attacks. The next chapter describes how to enable TLS.

Connect Securely

The Java bindings library lets you connect to a Daml Ledger via a secure connection. The builders created by `DamlLedgerClient.newBuilder` default to a plaintext connection, but you can invoke `withSslContext` to pass an `SslContext`. Using the default plaintext connection is useful only when connecting to a locally running Sandbox for development purposes.

Secure connections to a Daml Ledger must be configured to use client authentication certificates, which can be provided by a Ledger Operator.

For information on how to set up an `SslContext` with the provided certificates for client authentication, please consult the gRPC documentation on TLS with OpenSSL as well as the HelloWorldClientTls example of the `grpc-java` project.

Advanced Connection Settings

Sometimes the default settings for gRPC connections/channels are not suitable for a given situation. These use cases are supported by creating a custom NettyChannelBuilder object and passing the it to the `newBuilder` static method defined over DamlLedgerClient.

Example Projects

Example projects using the Java bindings are available on GitHub. *Read more about them here*.

Generate Java Code from Daml

Introduction

When writing applications for the ledger in Java, you want to work with a representation of Daml templates and data types in Java that closely resemble the original Daml code while still being as true to the native types in Java as possible. To achieve this, you can use Daml to Java code generator (Java codegen) to generate Java types based on a Daml model. You can then use these types in your Java code when reading information from and sending data to the ledger.

The *Daml assistant documentation* describes how to run and configure the code generator for all supported bindings, including Java.

The rest of this page describes Java-specific topics.

Understand the Generated Java Model

The Java codegen generates source files in a directory tree under the output directory specified on the command line.

Map Daml Primitives to Java Types

Daml built-in types are translated to the following equivalent types in Java:

Daml type	Java type	Java Bindings Value Type
`Int`	`java.lang.Long`	Int64
`Numeric`	`java.math.BigDecimal`	Numeric
`Text`	`java.lang.String`	Text
`Bool`	`java.util.Boolean`	Bool
`Party`	`java.lang.String`	Party
`Date`	`java.time.LocalDate`	Date
`Time`	`java.time.Instant`	Timestamp
`List` or `[]`	`java.util.List`	DamlList
`TextMap`	`java.util.Map` Restricted to using `String` keys.	DamlTextMap
`Optional`	`java.util.Optional`	DamlOptional
`()` (Unit)	**None** since the Java language doesn't have a direct equivalent of Daml's Unit type `()`, the generated code uses the Java Bindings value type.	Unit
`ContractId`	Fields of type `ContractId X` refer to the generated `ContractId` class of the respective template `X`.	ContractId

Understand Escaping Rules

To avoid clashes with Java keywords, the Java codegen applies escaping rules to the following Daml identifiers:

Type names (except the already mapped *built-in types*)
Constructor names
Type parameters
Module names
Field names

If any of these identifiers match one of the Java reserved keywords, the Java codegen appends a dollar sign `$` to the name. For example, a field with the name `import` will be generated as a Java field with the name `import$`.

Understand the Generated Classes

Every user-defined data type in Daml (template, record, and variant) is represented by one or more Java classes as described in this section.

The Java package for the generated classes is the equivalent of the lowercase Daml module name.

Listing 2: Daml

```
module Foo.Bar.Baz where
```

Listing 3: Java

```
package foo.bar.baz;
```

Records (a.k.a Product Types)

A *Daml record* is represented by a Java class with fields that have the same name as the Daml record fields. A Daml field having the type of another record is represented as a field having the type of the generated class for that record.

Listing 4: Com/Acme/ProductTypes.daml

```
module Com.Acme.ProductTypes where

data Person = Person with name : Name; age : Decimal
data Name = Name with firstName : Text; lastName : Text
```

A Java file is generated that defines the class for the type `Person`:

Listing 5: com/acme/producttypes/Person.java

```
package com.acme.producttypes;

public class Person extends DamlRecord<Person> {
  public final Name name;
  public final BigDecimal age;

  public static Person fromValue(Value value$) { /* ... */ }

  public Person(Name name, BigDecimal age) { /* ... */ }
  public DamlRecord toValue() { /* ... */ }
}
```

A Java file is generated that defines the class for the type `Name`:

Listing 6: com/acme/producttypes/Name.java

```
package com.acme.producttypes;

public class Name extends DamlRecord<Name> {
  public final String firstName;
  public final String lastName;
```

(continues on next page)

(continued from previous page)

```
  public static Person fromValue(Value value$) { /* ... */ }

  public Name(String firstName, String lastName) { /* ... */ }
  public DamlRecord toValue() { /* ... */ }
}
```

Templates

The Java codegen generates three classes for a Daml template:

TemplateName Represents the contract data or the template fields.

TemplateName.ContractId Used whenever a contract ID of the corresponding template is used in another template or record, for example: `data Foo = Foo (ContractId Bar)`. This class also provides methods to generate an `ExerciseCommand` for each choice that can be sent to the ledger with the Java Bindings.

TemplateName.Contract Represents an actual contract on the ledger. It contains a field for the contract ID (of type `TemplateName.ContractId`) and a field for the template data (of type `TemplateName`). With the static method `TemplateName.Contract.fromCreatedEvent`, you can deserialize a CreatedEvent to an instance of `TemplateName.Contract`.

Listing 7: Com/Acme/Templates.daml

```
module Com.Acme.Templates where

data BarKey =
  BarKey
    with
      p : Party
      t : Text

template Bar
  with
    owner: Party
    name: Text
  where
    signatory owner

    key BarKey owner name : BarKey
    maintainer key.p

    choice Bar_SomeChoice: Bool
      with
        aName: Text
      controller owner
      do return True
```

A file is generated that defines five Java classes and an interface:

1. `Bar`
2. `Bar.ContractId`
3. `Bar.Contract`
4. `Bar.CreateAnd`

5. Bar.ByKey
6. Bar.Exercises

Listing 8: com/acme/templates/Bar.java

```
package com.acme.templates;

public class Bar extends Template {

  public static final Identifier TEMPLATE_ID = new Identifier("some-package-id",
↪"Com.Acme.Templates", "Bar");

  public static final Choice<Bar, Archive, Unit> CHOICE_Archive =
    Choice.create(/* ... */);

  public static final ContractCompanion.WithKey<Contract, ContractId, Bar, BarKey>
↪ COMPANION =
      new ContractCompanion.WithKey<>("com.acme.templates.Bar",
        TEMPLATE_ID, ContractId::new, Bar::fromValue, Contract::new, e -> BarKey.
↪fromValue(e), List.of(CHOICE_Archive));

  public final String owner;
  public final String name;

  public CreateAnd createAnd() { /* ... */ }

  public static ByKey byKey(BarKey key) { /* ... */ }

  public static class ContractId extends com.daml.ledger.javaapi.data.codegen.
↪ContractId<Bar>
      implements Exercises<ExerciseCommand> {
    // inherited:
    public final String contractId;
  }

  public interface Exercises<Cmd> extends com.daml.ledger.javaapi.data.codegen.
↪Exercises<Cmd> {
    default Cmd exerciseArchive(Unit arg) { /* ... */ }

    default Cmd exerciseBar_SomeChoice(Bar_SomeChoice arg) { /* ... */ }

    default Cmd exerciseBar_SomeChoice(String aName) { /* ... */ }
  }

  public static class Contract extends ContractWithKey<ContractId, Bar, BarKey> {
    // inherited:
    public final ContractId id;
    public final Bar data;

    public static Contract fromCreatedEvent(CreatedEvent event) { /* ... */ }
  }

  public static final class CreateAnd
      extends com.daml.ledger.javaapi.data.codegen.CreateAnd
      implements Exercises<CreateAndExerciseCommand> { /* ... */ }

  public static final class ByKey
```

(continues on next page)

(continued from previous page)

```
      extends com.daml.ledger.javaapi.data.codegen.ByKey
      implements Exercises<ExerciseByKeyCommand> { /* ... */ }
}
```

Note that `byKey` and `ByKey` will only be generated for templates that define a key.

Variants (a.k.a Sum Types)

A *variant or sum type* is a type with multiple constructors, where each constructor wraps a value of another type. The generated code is comprised of an abstract class for the variant type itself and a subclass thereof for each constructor. Classes for variant constructors are similar to classes for records.

Listing 9: Com/Acme/Variants.daml

```
module Com.Acme.Variants where

data BookAttribute = Pages Int
                   | Authors [Text]
                   | Title Text
                   | Published with year: Int; publisher: Text
```

The Java code generated for this variant is:

Listing 10: com/acme/variants/BookAttribute.java

```
package com.acme.variants;

public class BookAttribute extends Variant<BookAttribute> {
  public static BookAttribute fromValue(Value value) { /* ... */ }

  public static BookAttribute fromValue(Value value) { /* ... */ }
  public abstract Variant toValue();
}
```

Listing 11: com/acme/variants/bookattribute/Pages.java

```
package com.acme.variants.bookattribute;

public class Pages extends BookAttribute {
  public final Long longValue;

  public static Pages fromValue(Value value) { /* ... */ }

  public Pages(Long longValue) { /* ... */ }
  public Variant toValue() { /* ... */ }
}
```

Listing 12: com/acme/variants/bookattribute/Authors.java

```
package com.acme.variants.bookattribute;
```

(continues on next page)

(continued from previous page)

```
public class Authors extends BookAttribute {
  public final List<String> listValue;

  public static Authors fromValue(Value value) { /* ... */ }

  public Author(List<String> listValue) { /* ... */ }
  public Variant toValue() { /* ... */ }

}
```

Listing 13: com/acme/variants/bookattribute/Title.java

```
package com.acme.variants.bookattribute;

public class Title extends BookAttribute {
  public final String stringValue;

  public static Title fromValue(Value value) { /* ... */ }

  public Title(String stringValue) { /* ... */ }
  public Variant toValue() { /* ... */ }
}
```

Listing 14: com/acme/variants/bookattribute/Published.java

```
package com.acme.variants.bookattribute;

public class Published extends BookAttribute {
  public final Long year;
  public final String publisher;

  public static Published fromValue(Value value) { /* ... */ }

  public Published(Long year, String publisher) { /* ... */ }
  public Variant toValue() { /* ... */ }
}
```

Enums

An enum type is a simplified *sum type* with multiple constructors but without argument nor type parameters. The generated code is standard java Enum whose constants map enum type constructors.

Listing 15: Com/Acme/Enum.daml

```
module Com.Acme.Enum where

data Color = Red | Blue | Green
```

The Java code generated for this variant is:

Listing 16: com/acme/enum/Color.java

```
package com.acme.enum;

public enum Color implements DamlEnum<Color> {
  RED,
  GREEN,
  BLUE;

  /* ... */
  public static final Color fromValue(Value value$) { /* ... */ }
  public final DamlEnum toValue() { /* ... */ }
}
```

Parameterized Types

Note: This section is only included for completeness. The `fromValue` and `toValue` methods would typically come from a template that doesn't have any unbound type parameters.

The Java codegen uses Java Generic types to represent *Daml parameterized types*.

This Daml fragment defines the parameterized type `Attribute`, used by the `BookAttribute` type for modeling the characteristics of the book:

Listing 17: Com/Acme/ParameterizedTypes.daml

```
module Com.Acme.ParameterizedTypes where

data Attribute a = Attribute
    with v : a

data BookAttributes = BookAttributes with
   pages : (Attribute Int)
   authors : (Attribute [Text])
   title : (Attribute Text)
```

The Java codegen generates a Java file with a generic class for the `Attribute a` data type:

Listing 18: com/acme/parameterizedtypes/Attribute.java

```
package com.acme.parameterizedtypes;

public class Attribute<a> {
  public final a value;

  public Attribute(a value) { /* ... */  }

  public DamlRecord toValue(Function<a, Value> toValuea) { /* ... */ }

  public static <a> Attribute<a> fromValue(Value value$, Function<Value, a>
↪fromValuea) { /* ... */ }
}
```

Convert a Value of a Generated Type to a Java Bindings Value

To convert an instance of the generic type `Attribute<a>` to a Java Bindings Value, call the `toValue` method and pass a function as the `toValuea` argument for converting the field of type `a` to the respective Java Bindings Value. The name of the parameter consists of `toValue` and the name of the type parameter, in this case `a`, to form the name `toValuea`.

Below is a Java fragment that converts an attribute with a `java.lang.Long` value to the Java Bindings representation using the *method reference* `Int64::new`.

```
Attribute<Long> pagesAttribute = new Attributes<>(42L);

Value serializedPages = pagesAttribute.toValue(Int64::new);
```

See *Daml To Java Type Mapping* for an overview of the Java Bindings Value types.

Note: If the Daml type is a record or variant with more than one type parameter, you need to pass a conversion function to the `toValue` method for each type parameter.

Create a Value of a Generated Type from a Java Bindings Value

Analogous to the `toValue` method, to create a value of a generated type, call the method `fromValue` and pass conversion functions from a Java Bindings Value type to the expected Java type.

```
Attribute<Long> pagesAttribute = Attribute.<Long>fromValue(serializedPages,
    f -> f.asInt64().getOrElseThrow(() -> throw new IllegalArgumentException(
↪"Expected Int field").getValue());
```

See Java Bindings Value class for the methods to transform the Java Bindings types into corresponding Java types.

Non-exposed Parameterized Types

If the parameterized type is contained in a type where the *actual* type is specified (as in the `BookAttributes` type above), then the conversion methods of the enclosing type provides the required conversion function parameters automatically.

Convert Optional Values

The conversion of the Java `Optional` requires two steps. The `Optional` must be mapped in order to convert its contains before to be passed to `DamlOptional::of` function.

```
Attribute<Optional<Long>> idAttribute = new Attribute<List<Long>>(Optional.
↪of(42));

val serializedId = DamlOptional.of(idAttribute.map(Int64::new));
```

To convert back DamlOptional to Java `Optional`, one must use the containers method `toOptional`. This method expects a function to convert back the value possibly contains in the container.

```
Attribute<Optional<Long>> idAttribute2 =
  serializedId.toOptional(v -> v.asInt64().orElseThrow(() -> new
↪IllegalArgumentException("Expected Int64 element")));
```

Convert Collection Values

DamlCollectors provides collectors to converted Java collection containers such as `List` and `Map` to DamlValues in one pass. The builders for those collectors require functions to convert the element of the container.

```
Attribute<List<String>> authorsAttribute =
    new Attribute<List<String>>(Arrays.asList("Homer", "Ovid", "Vergil"));

Value serializedAuthors =
    authorsAttribute.toValue(f -> f.stream().collect(DamlCollector.
↪toList(Text::new));
```

To convert back Daml containers to Java ones, one must use the containers methods `toList` or `toMap`. Those methods expect functions to convert back the container's entries.

```
Attribute<List<String>> authorsAttribute2 =
    Attribute.<List<String>>fromValue(
        serializedAuthors,
        f0 -> f0.asList().orElseThrow(() -> new IllegalArgumentException(
↪"Expected DamlList field"))
            .toList(
                f1 -> f1.asText().orElseThrow(() -> new IllegalArgumentException(
↪"Expected Text element"))
                      .getValue()
            )
    );
```

Daml Interfaces

From this daml definition:

Listing 19: Interfaces.daml

```
module Interfaces where

data TIfView = TIfView { name : Text }

interface TIf where
  viewtype TIfView
  getOwner: Party
  dup: Update (ContractId TIf)
  choice Ham: ContractId TIf with
    controller getOwner this
    do dup this
  choice Useless: ContractId TIf with
      interfacely: ContractId TIf
    controller getOwner this
```

(continues on next page)

(continued from previous page)

```
    do
      dup this

template Child
    with
    party: Party
  where
    signatory party
    choice Bar: () with
      controller party
      do
        return ()

    interface instance TIf for Child where
      view = TIfView "Child"
      getOwner = party
      dup = toInterfaceContractId <$> create this
```

The generated file for the interface definition can be seen below. Effectively it is a class that contains only the inner type ContractId because one will always only be able to deal with Interfaces via their ContractId.

Listing 20: interfaces/TIf.java

```
package interfaces

/* imports */

public final class TIf {
  public static final Identifier TEMPLATE_ID = new Identifier(
↪"94fb4fa48cef1ec7d474ff3d6883a00b2f337666c302ec5e2b87e986da5c27a3", "Interfaces
↪", "TIf");

  public static final Choice<TIf, Transfer, ContractId> CHOICE_Transfer =
    Choice.create(/* ... */);

  public static final Choice<TIf, Archive, Unit> CHOICE_Archive =
    Choice.create(/* ... */);

  public static final INTERFACE INTERFACE = new INTERFACE();

  public static final class ContractId extends com.daml.ledger.javaapi.data.
↪codegen.ContractId<TIf>
      implements Exercises<ExerciseCommand> {
    public ContractId(String contractId) { /* ... */ }
  }

  public interface Exercises<Cmd> extends com.daml.ledger.javaapi.data.codegen.
↪Exercises<Cmd> {
    default Cmd exerciseUseless(Useless arg) { /* ... */ }

    default Cmd exerciseHam(Ham arg) { /* ... */ }
  }

  public static final class CreateAnd
```

(continues on next page)

(continued from previous page)

```
      extends com.daml.ledger.javaapi.data.codegen.CreateAnd.ToInterface
      implements Exercises<CreateAndExerciseCommand> { /* ... */ }

  public static final class ByKey
      extends com.daml.ledger.javaapi.data.codegen.ByKey.ToInterface
      implements Exercises<ExerciseByKeyCommand> { /* ... */ }

  public static final class INTERFACE extends InterfaceCompanion<TIf> { /* ... */}
}
```

For templates the code generation will be slightly different if a template implements interfaces. To allow converting the ContractId of a template to an interface ContractId, an additional conversion method called *toInterface* is generated. An `unsafeFromInterface` is also generated to make the [unchecked] conversion in the other direction.

Listing 21: interfaces/Child.java

```
package interfaces

/* ... */

public final class Child extends Template {

  /* ... */

  public static final class ContractId extends com.daml.ledger.javaapi.data.
↪codegen.ContractId<Child>
      implements Exercises<ExerciseCommand> {

    /* ... */

    public TIf.ContractId toInterface(TIf.INTERFACE interfaceCompanion) { /* ...
↪*/ }

    public static ContractId unsafeFromInterface(TIf.ContractId
↪interfaceContractId) { /* ... */ }

  }

  public interface Exercises<Cmd> extends com.daml.ledger.javaapi.data.codegen.
↪Exercises<Cmd> {
    default Cmd exerciseBar(Bar arg) { /* ... */ }

    default Cmd exerciseBar() { /* ... */ }
  }

  /* ... */

}
```

Java Bindings Example Project

To try out the Java bindings library, use the examples on GitHub: `PingPongReactive`.

The example implements the `PingPong` application, which consists of:

a Daml model with two contract templates, `Ping` and `Pong`
two parties, `Alice` and `Bob`

The logic of the application goes like this:

1. The application injects a contract of type `Ping` for `Alice`.
2. `Alice` sees this contract and exercises the consuming choice `RespondPong` to create a contract of type `Pong` for `Bob`.
3. `Bob` sees this contract and exercises the consuming choice `RespondPing` to create a contract of type `Ping` for `Alice`.
4. Points 2 and 3 are repeated until the maximum number of contracts defined in the Daml is reached.

Set Up the Example Projects

To set up the example projects, clone the public GitHub repository at github.com/digital-asset/ex-java-bindings and follow the setup instruction in the README file.

This project contains two examples of the PingPong application, built directly with gRPC and using the RxJava2-based Java bindings.

Example Project

PingPongMain.java

The entry point for the Java code is the main class `src/main/java/examples/pingpong/grpc/PingPongMain.java`. Look at this class to see:

how to connect to and interact with a Daml Ledger via the Java bindings
how to use the Reactive layer to build an automation for both parties.

At high level, the code does the following steps:

creates an instance of `DamlLedgerClient` connecting to an existing Ledger
connect this instance to the Ledger with `DamlLedgerClient.connect()`
create two instances of `PingPongProcessor`, which contain the logic of the automation
(This is where the application reacts to the new `Ping` or `Pong` contracts.)
run the `PingPongProcessor` forever by connecting them to the incoming transactions
inject some contracts for each party of both templates
wait until the application is done

PingPongProcessor.runIndefinitely()

The core of the application is the `PingPongProcessor.runIndefinitely()`.

The `PingPongProcessor` queries the transactions first via the `TransactionsClient` of the `DamlLedgerClient`. Then, for each transaction, it produces `Commands` that will be sent to the Ledger via the `CommandSubmissionClient` of the `DamlLedgerClient`.

Output

The application prints statements similar to these:

```
Bob is exercising RespondPong on #1:0 in workflow Ping-Alice-1 at count 0
Alice is exercising RespondPing on #344:1 in workflow Ping-Alice-7 at count 9
```

The first line shows that:

> `Bob` is exercising the `RespondPong` choice on the contract with ID `#1:0` for the workflow `Ping-Alice-1`.
> Count `0` means that this is the first choice after the initial `Ping` contract.
> The workflow ID `Ping-Alice-1` conveys that this is the workflow triggered by the second initial `Ping` contract that was created by `Alice`.

The second line is analogous to the first one.

Daml IOU Quickstart Tutorial

In this guide, you will learn about developer tools and Daml applications by:

> developing a simple ledger application for issuing, managing, transferring and trading IOUs (I Owe You!)
> developing an integration layer that exposes some of the functionality via custom REST services

Prerequisites:

> You understand what an IOU is. If you are not sure, read the *IOU tutorial overview*.
> You have installed the SDK. See *installation*.

Download the Quickstart Application

You can get the quickstart application using the Daml assistant (`daml`):

1. Run `daml new quickstart --template quickstart-java`
   This creates the `quickstart-java` application into a new folder called `quickstart`.
2. Run `cd quickstart` to change into the new directory.

Folder Structure

The project contains the following files:

```
.
├── daml
│   ├── Iou.daml
│   ├── IouTrade.daml
│   ├── Main.daml
│   └── Tests
│       ├── Iou.daml
│       └── Trade.daml
├── daml.yaml
├── frontend-config.js
├── pom.xml
└── src
    └── main
        ├── java
        │   └── com
        │       └── daml
        │           └── quickstart
        │               └── iou
        │                   └── IouMain.java
        └── resources
            └── logback.xml
```

`daml.yaml` is a Daml project config file used by the SDK to find out how to build the Daml project and how to run it.
`daml` contains the *Daml code* specifying the contract model for the ledger.
`daml/Tests` contains *test scripts* for the Daml model.
`frontend-config.js` is a configuration file for the *Navigator* frontend.
`pom.xml` and `src/main/java` constitute a *Java application* that provides REST services to interact with the ledger.

You will explore these in more detail through the rest of this guide.

Understand IOUs

To run through this guide, you will need to understand what an IOU is. This section describes the properties of an IOU like a bank bill that make it useful as a representation and transfer of value.

A bank bill represents a contract between the owner of the bill and its issuer, the central bank. Historically, it is a bearer instrument - it gives anyone who holds it the right to demand a fixed amount of material value, often gold, from the issuer in exchange for the note.

To do this, the note must have certain properties. In particular, the British pound note shown below illustrates the key elements that are needed to describe money in Daml:

1) The Legal Agreement

For a long time, money was backed by physical gold or silver stored in a central bank. The British pound note, for example, represented a promise by the central bank to provide a certain amount of gold or silver in exchange for the note. This historical artifact is still represented by the following statement:


```
I promise to pay the bearer on demand the sum of five pounds.
```

The true value of the note comes from the fact that it physically represents a bearer right that is matched by an obligation on the issuer.

2) The Signature of the Counterparty

The value of a right described in a legal agreement is based on a matching obligation for a counterparty. The British pound note would be worthless if the central bank, as the issuer, did not recognize its obligation to provide a certain amount of gold or silver in exchange for the note. The chief cashier confirms this obligation by signing the note as a delegate for the Bank of England. In general, determining the parties that are involved in a contract is key to understanding its true value.

3) The Security Token

Another feature of the pound note is the security token embedded within the physical paper. It allows the note to be authenticated with limited effort by holding it against a light source. Even a third party can verify the note without requiring explicit confirmation from the issuer that it still acknowledges the associated obligations.

4) The Unique Identifier

Every note has a unique registration number that allows the issuer to track their obligations and detect duplicate bills. Once the issuer has fulfilled the obligations associated with a particular note, duplicates with the same identifier automatically become invalid.

5) The Distribution Mechanism

The note itself is printed on paper, and its legal owner is the person holding it. The physical form of the note allows the rights associated with it to be transferred to other parties that are not explicitly mentioned in the contract.

Run the Application Using Prototyping Tools

In this section, you will run the quickstart application and get introduced to the main tools for prototyping Daml:

1. To compile the Daml model, run `daml build`
   This creates a *DAR file* (DAR is just the format that Daml compiles to) called `.daml/dist/quickstart-0.0.1.dar`. The output should look like this:

```
2022-09-08 14:33:41.65 [INFO]  [build]
Compiling quickstart to a DAR.

2022-09-08 14:33:42.90 [INFO]  [build]
Created .daml/dist/quickstart-0.0.1.dar
```

2. To run the *sandbox* (a lightweight local version of the ledger), run:

```
daml sandbox --port 6865
```

3. In a separate terminal run the following:

   Upload the DAR file:

```
daml ledger upload-dar --host localhost --port 6865 .daml/dist/quickstart-0.0.
↪1.dar
```

   Run the init script:

```
daml script --ledger-host localhost --ledger-port 6865 --dar .daml/dist/
↪quickstart-0.0.1.dar --script-name Main:initialize --output-file output.json
```

   Start the *Navigator*, a browser-based ledger front-end, by running:

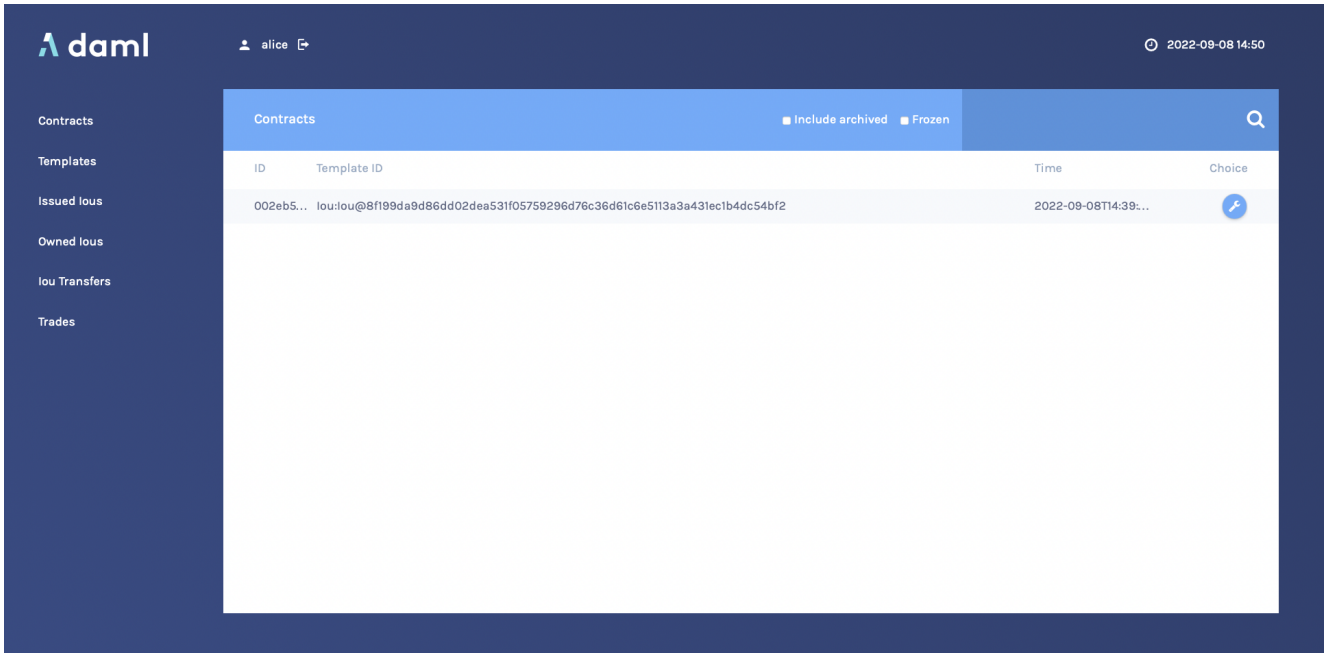
```
daml navigator server localhost 6865 --port 7500
```

   The Navigator automatically connects to the sandbox. You can access it on port `7500`.

Try the Application

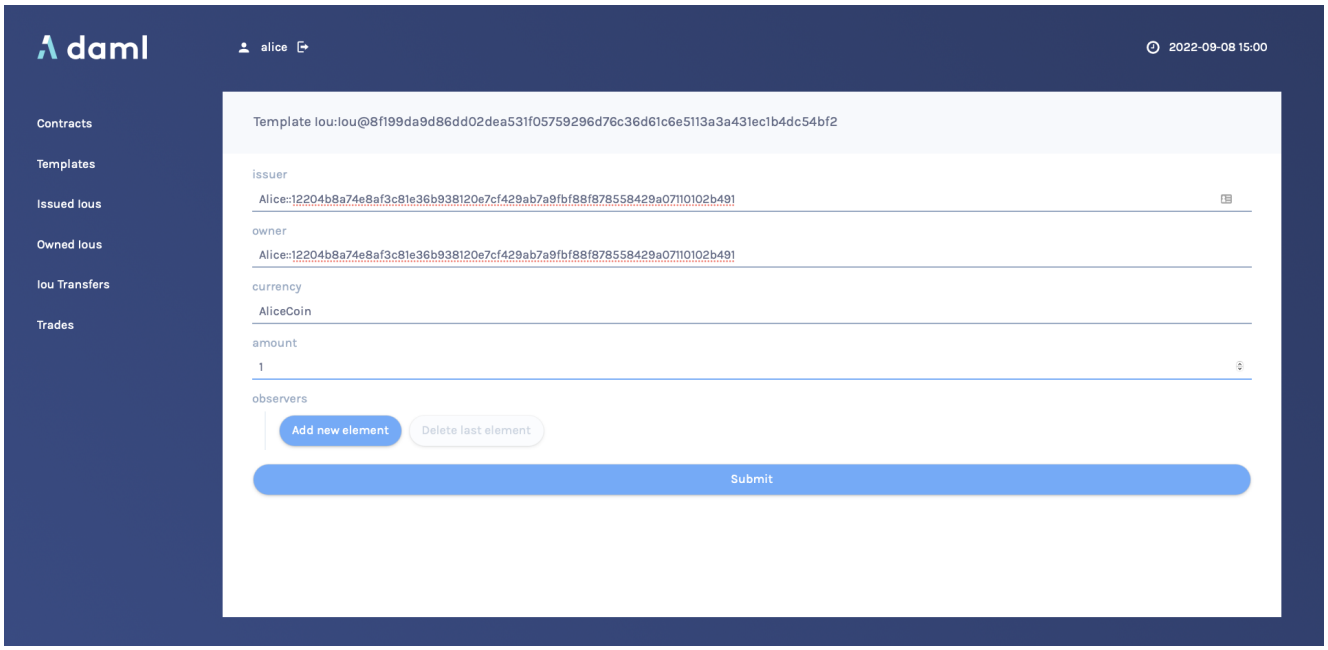
Now everything is running, you can try out the quickstart application:

1. Go to http://localhost:7500/. This is the *Navigator*, which you launched *earlier*.
2. On the login screen, select `alice` from the dropdown. This logs you in as `alice`.
   This takes you to the contracts view:
   This is showing you what contracts are currently active on the sandbox ledger and visible to `alice`. You can see that there is a single such contract, in our case with Id `002eb5...`, created from a *template* called `Iou:Iou@8f199da...`.
   Your contract ID will vary. The actual value doesn't matter. We'll refer to this contract as `002eb5` in the rest of this document, and you'll need to substitute your own value mentally.
3. On the left-hand side, you can see what the pages the Navigator contains:
   - Contracts
   - Templates
   - Issued Ious
   - Owned Ious
   - Iou Transfers
   - Trades

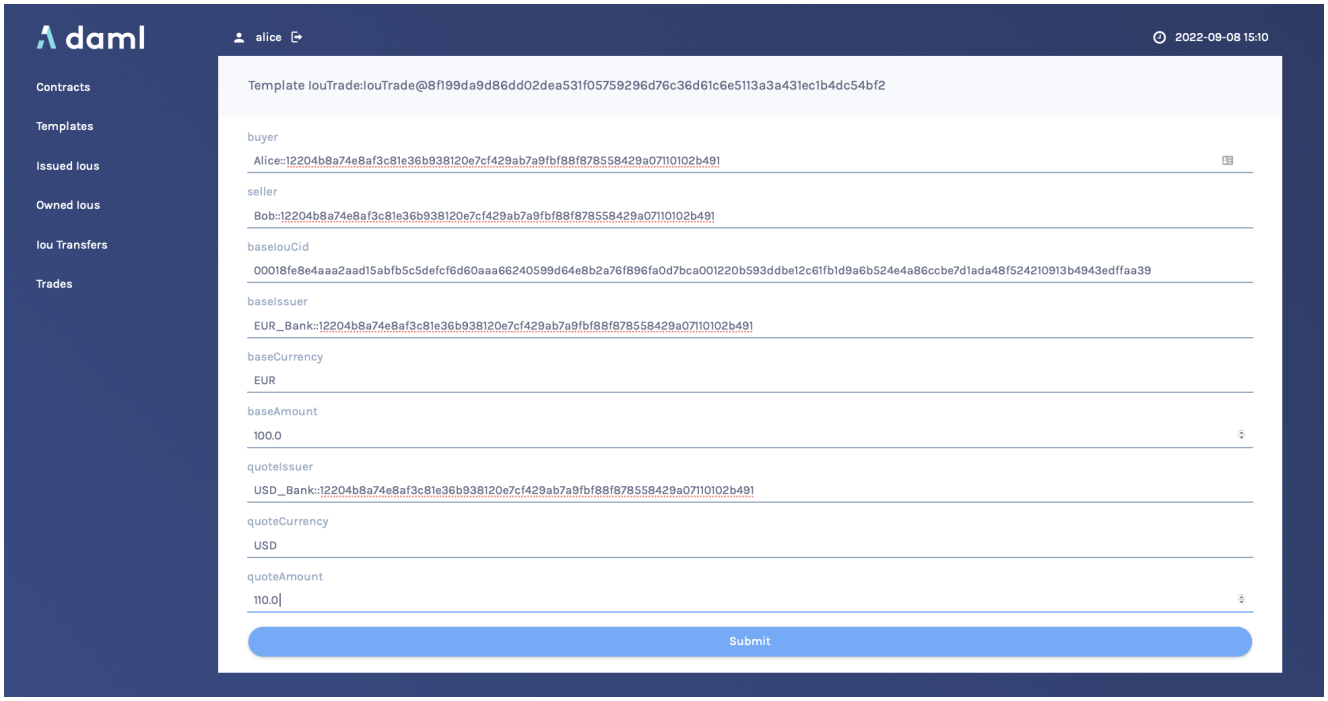


Contracts and **Templates** are standard views, available in any application. The others are created just for this application, specified in the `frontend-config.js` file.
For information on creating custom Navigator views, see *Customizable table views*.

4. Click **Templates** to open the Templates page.
   This displays all available *contract templates*. Instances of contracts (or just *contracts*) are created from these templates. The names of the templates are of the format *module:template@hash*. Including the hash disambiguates templates, even when identical module and template names are used between packages.
   On the far right, you see the number of *contracts* that you can see for each template, if any, or – for no contract .
5. Try creating a contract from a template. Issue an Iou to yourself by clicking on the `Iou:Iou@8f199...` row, filling it out as shown below (use the provided auto-complete feature for the `Party` values in `issuer` and `owner`) and clicking **Submit**.

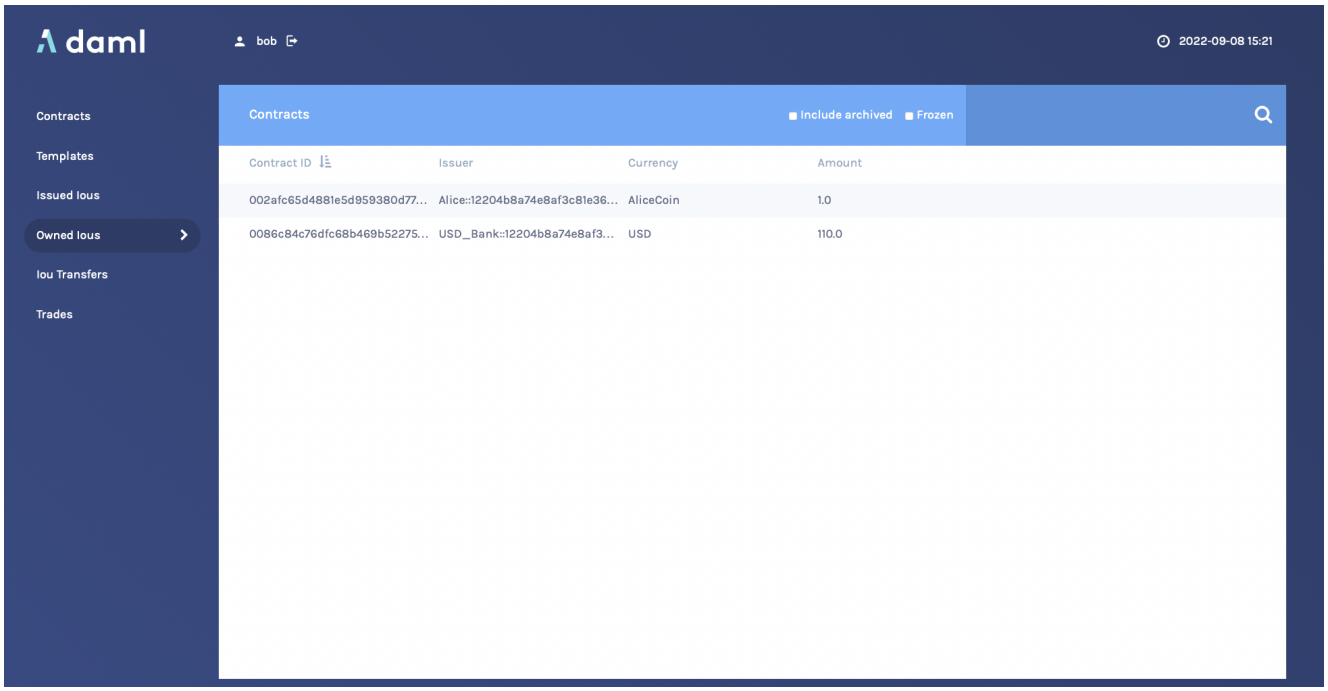


6. On the left-hand side, click **Issued Ious** to go to that page. You can see the Iou you just issued yourself.
7. Now, try transferring this Iou to someone else. Click on your Iou, select `Iou_Transfer`, select `Bob::...` as the new owner and hit **Submit**.
8. Go to the **Owned Ious** page.
   The screen shows the same contract `002eb5` that you already saw on the *Contracts* page. It is an Iou for 100, issued by `EUR_Bank::...`.
9. Go to the **Iou Transfers** page. It shows the transfer of your recently issued Iou to Bob, but Bob has not accepted the transfer, so it is not settled.
   This is an important part of Daml: nobody can be forced into owning an *Iou*, or indeed agreeing to any other contract. They must explicitly consent.
   You could cancel the transfer by using the `IouTransfer_Cancel` choice within it, but for this walk-through, leave it alone for the time being.
10. Try asking *Bob* to exchange your 100 for $110. To do so, you first have to show your Iou to *Bob* so that he can verify the settlement transaction, should he accept the proposal.
    Go back to **Owned Ious**, open the Iou for 100 and click on the button `Iou_AddObserver`. Select `Bob::...` as the `newObserver`.
    Contracts in Daml are immutable, meaning they cannot be changed, only created and archived. If you head back to the **Owned Ious** screen, you can see that the Iou now has a new Contract ID. In our case, it's `00018fe...`.
11. To propose the trade, go to the **Templates** screen. Click on the `IouTrade:IouTrade@...` template, fill in the form as shown below and submit the transaction. Remember to use the dropdown for the values of `buyer`, `seller`, `baseIouCid`, `baseIssuer`, and `quoteIssuer`.



12. Go to the **Trades** page. It shows the just-proposed trade.
13. You are now going to switch user to Bob, so you can accept the trades you have just proposed. Start by clicking on the logout button next to the username, at the top of the screen. On the login page, select `bob` from the dropdown.
14. First, accept the transfer of the *AliceCoin*. Go to the **Iou Transfers** page, click on the row of the transfer, and click `IouTransfer_Accept`, then **Submit**.
15. Go to the **Owned Ious** page. It now shows the *AliceCoin*.
    It also shows an *Iou* for $110 issued by `USD_Bank::...`. This matches the trade proposal

you made earlier as Alice. Remember the first few characters of its Contract ID (in our case `0086c84`).



16. Settle the trade. Go to the **Trades** page, and click on the row of the proposal. Accept the trade by clicking `IouTrade_Accept`. In the popup, select the Contract ID you just noted from the dropdown as the `quoteIouCid`, then click **Submit**.
    The two legs of the transfer are now settled atomically in a single transaction. The trade either fails or succeeds as a whole.
17. Privacy is an important feature of Daml. You can check that Alice and Bob's privacy relative to the Banks was preserved.
    To do this, log out, then log in as `us`, which maps to `USD_Bank::...`.
    On the **Contracts** page, select **Include archived**. The page now shows all the contracts that `USD_Bank::...` has ever known about.
    There are just five contracts:
    Three contracts created on startup:
    1. A self-issued *Iou* for $110.
    2. The *IouTransfer* to transfer that *Iou* to Bob
    3. The resulting *Iou* owned by Bob.

    The transfer of Bob's *Iou* to Alice that happened as part of the trade. Note that this is a transient contract that got archived in the same transaction it got created in.
    The new $110 *Iou* owned by Alice. This is the only active contract.

    Importantly, `USD_Bank::...` does not know anything about the trade or the EUR-leg. It has no idea what was exchanged for those $110, or indeed if anything was exchanged at all. For more information on privacy, refer to the *Daml Ledger Model*.

Note: `USD_Bank::...` does know about an intermediate *IouTransfer* contract that was created and consumed as part of the atomic settlement in the previous step. Since that contract was never active on the ledger, it is not shown in Navigator. You will see how to view a complete transaction graph, including who knows what, in *Test Using Daml Script* below.

Get Started with Daml

The *contract model* specifies the possible contracts, as well as the allowed transactions on the ledger, and is written in Daml.

The core concept in Daml is a *contract template* - you used them earlier to create contracts. Contract templates specify:

a type of contract that may exist on the ledger, including a corresponding data type
the *signatories*, who need to agree to the *creation* of a contract of that type
the *rights* or *choices* given to parties by a contract of that type
constraints or conditions on the data on a contract
additional parties, called observers, who can see the contract

For more information about Daml Ledgers, consult *Daml Ledger Model* for an in-depth technical description.

Develop with Daml Studio

Take a look at the Daml that specifies the contract model in the quickstart application. The core template is `Iou`.

1. Open *Daml Studio*, a Daml IDE based on VS Code, by running `daml studio` from the root of your project.
2. Using the explorer on the left, open `daml/Iou.daml`.

The first (uncommented, non-empty) line specifies the module name:

```
module Iou where
```

Next, a template called *Iou* is declared together with its datatype. This template has five fields:

```
template Iou
  with
    issuer : Party
    owner : Party
    currency : Text
    amount : Decimal
    observers : [Party]
```

Conditions for the creation of a contract are specified using the *ensure* and *signatory* keywords:

```
    ensure amount > 0.0

    signatory issuer, owner
```

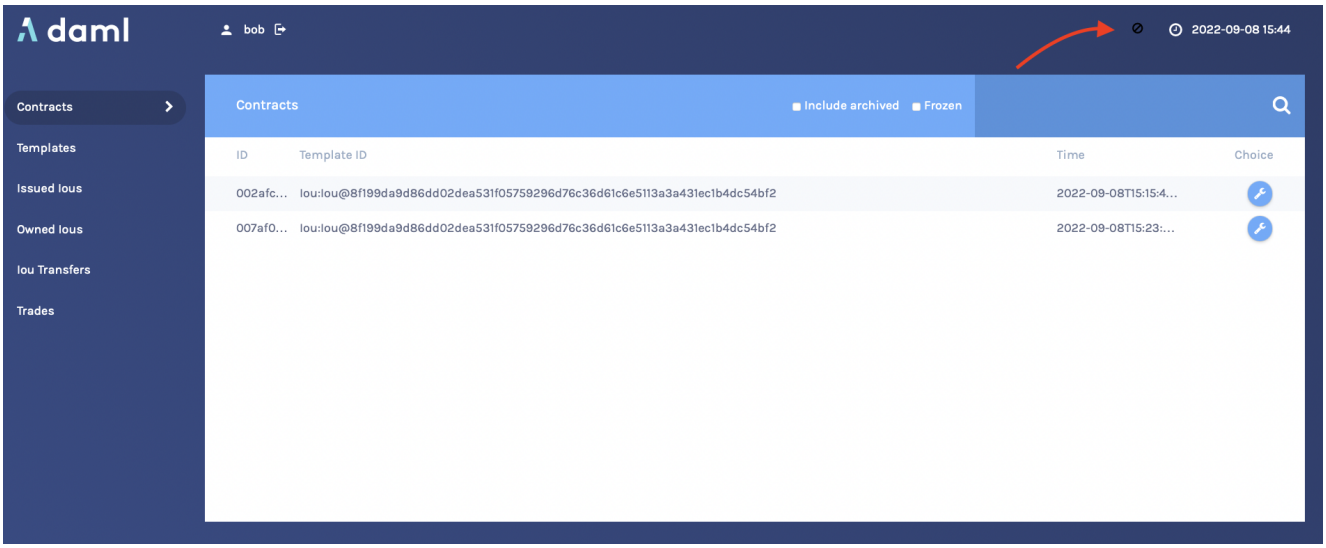
In this case, there are two conditions:

An `Iou` can only be created if it is authorized by both `issuer` and `owner`.
The `amount` needs to be positive.

Earlier, as Alice, you authorized the creation of an `Iou`. The `amount` was `1.0`, and Alice was both `issuer` and `owner`, so both conditions were satisfied, and you could successfully create the contract.

To see this in action, go back to the Navigator and try to create the same `Iou` again, but with Bob as `owner` (with Alice as issuer). It will not work. Note that the Navigator shows success an failures as a

small icon in the top right, as highlighted here (it would be a small v for success):



Observers are specified using the `observer` keyword:

```
    observer observers
```

Here, `observer` is the keyword and `observers` refers to the field of the template.

Next, the *rights* or *choices* are defined, in this case with `owner` as the `controller`:

```
    choice Iou_Split : (IouCid, IouCid)
        with
        splitAmount: Decimal
      controller owner
      do
        let restAmount = amount - splitAmount
        splitCid <- create this with amount = splitAmount
        restCid <- create this with amount = restAmount
        return (splitCid, restCid)
```

```
    choice Iou_Merge : IouCid
      with
        otherCid: IouCid
      controller owner
      do
        otherIou <- fetch otherCid
        -- Check the two IOU's are compatible
        assert (
          currency == otherIou.currency &&
          owner == otherIou.owner &&
          issuer == otherIou.issuer
          )
        -- Retire the old Iou
        archive otherCid
        -- Return the merged Iou
        create this with amount = amount + otherIou.amount
```

```
    choice Iou_Transfer : ContractId IouTransfer
      with
```

(continues on next page)

(continued from previous page)

```
      newOwner : Party
    controller owner
    do create IouTransfer with iou = this; newOwner
```

```
  choice Iou_AddObserver : IouCid
    with
      newObserver : Party
    controller owner
    do create this with observers = newObserver :: observers

  choice Iou_RemoveObserver : IouCid
    with
      oldObserver : Party
    controller owner
    do create this with observers = filter (/= oldObserver) observers
```

Thus, `owner` has the right to:

Split the Iou.
Merge it with another one differing only on `amount`.
Initiate a transfer.
Add and remove observers.

The `Iou_Transfer` choice above takes a parameter called `newOwner` and creates a new `IouTransfer` contract and returns its `ContractId`. It is important to know that, by default, choices *consume* the contract on which they are exercised. Consuming, or archiving, makes the contract no longer active. So the `IouTransfer` replaces the `Iou`.

A more interesting choice is `IouTrade_Accept`. To look at it, open `IouTrade.daml`.

```
  choice IouTrade_Accept : (IouCid, IouCid)
    with
      quoteIouCid : IouCid
    controller seller
    do
      baseIou <- fetch baseIouCid
      baseIssuer === baseIou.issuer
      baseCurrency === baseIou.currency
      baseAmount === baseIou.amount
      buyer === baseIou.owner
      quoteIou <- fetch quoteIouCid
      quoteIssuer === quoteIou.issuer
      quoteCurrency === quoteIou.currency
      quoteAmount === quoteIou.amount
      seller === quoteIou.owner
      quoteIouTransferCid <- exercise quoteIouCid Iou_Transfer with
        newOwner = buyer
      transferredQuoteIouCid <- exercise quoteIouTransferCid IouTransfer_Accept
      baseIouTransferCid <- exercise baseIouCid Iou_Transfer with
        newOwner = seller
      transferredBaseIouCid <- exercise baseIouTransferCid IouTransfer_Accept
      return (transferredQuoteIouCid, transferredBaseIouCid)
```

This choice uses the === operator from the *Daml Standard Library* to check pre-conditions. The standard library is imported using `import DA.Assert` at the top of the module.

Then, it *composes* the `Iou_Transfer` and `IouTransfer_Accept` choices to build one big transaction. In this transaction, `buyer` and `seller` exchange their Ious atomically, without disclosing the entire transaction to all parties involved.

The *Issuers* of the two Ious, which are involved in the transaction because they are signatories on the `Iou` and `IouTransfer` contracts, only get to see the sub-transactions that concern them, as we saw earlier.

For a deeper introduction to Daml, consult the *Daml Reference*.

Test Using Daml Script

You can check the correct authorization and privacy of a contract model using *scripts*: tests that are written in Daml.

Scripts are a linear sequence of transactions that is evaluated using the same consistency, conformance and authorization rules as it would be on the full ledger server or the sandbox ledger. They are integrated into Daml Studio, which can show you the resulting transaction graph, making them a powerful tool to test and troubleshoot the contract model.

To take a look at the scripts in the quickstart application, open `daml/Tests/Trade.daml` in Daml Studio.

A script test is defined with `trade_test = script do`. The `submit` function takes a submitting party and a transaction, which is specified the same way as in contract choices.

The following block, for example, issues an `Iou` and transfers it to Alice:

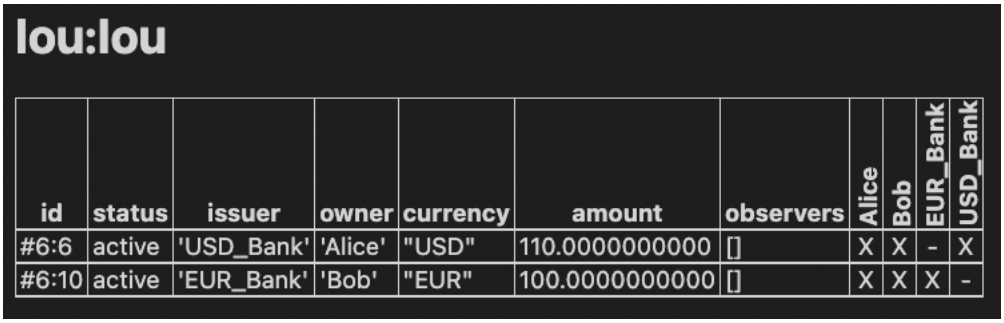
```
-- Banks issue IOU transfers.
iouTransferAliceCid <- submit eurBank do
  createAndExerciseCmd
    Iou with
      issuer = eurBank
      owner = eurBank
      currency = "EUR"
      amount = 100.0
      observers = []
    Iou_Transfer with
      newOwner = alice
```

Compare the script with the `initialize` script in `daml/Main.daml`. You will see that the script you used to initialize the sandbox is an initial segment of the `trade_test` script. The latter adds transactions to perform the trade you performed through Navigator, and a couple of transactions in which expectations are verified.

After a short time, the text *Script results* should appear above the test. Click on it (in `daml/Tests/Trade.daml`) to open the visualization of the resulting ledger state.

Each row shows a contract on the ledger. The last four columns show which parties know of which contracts. The remaining columns show the data on the contracts. You can see past contracts by checking the **Show archived** box at the top. Click the adjacent **Show transaction view** button to switch to a view of the entire transaction tree.

In the transaction view, transaction `6` is of particular interest, as it shows how the Ious are exchanged atomically in one transaction. The lines starting `disclosed to (since)` show that the Banks do indeed not know anything they should not:



Iou:Iou

id	status	issuer	owner	currency	amount	observers	Alice	Bob	EUR_Bank	USD_Bank
#6:6	active	'USD_Bank'	'Alice'	"USD"	110.0000000000	[]	X	X	-	X
#6:10	active	'EUR_Bank'	'Bob'	"EUR"	100.0000000000	[]	X	X	X	-

```
TX 6 1970-01-01T00:00:00Z (Tests.Trade:70:14)
#6:0
│   disclosed to (since): 'Alice' (6), 'Bob' (6)
└─> 'Bob' exercises IouTrade_Accept on #5:0 (IouTrade:IouTrade)
           with
           quoteIouCid = #3:1
      children:
      #6:1
      │   disclosed to (since): 'Alice' (6), 'Bob' (6), 'EUR_Bank' (6)
      └─> 'Alice' and 'EUR_Bank' fetch #4:1 (Iou:Iou)

      #6:2
      │   disclosed to (since): 'Alice' (6), 'Bob' (6), 'USD_Bank' (6)
      └─> 'Bob' and 'USD_Bank' fetch #3:1 (Iou:Iou)

      #6:3
      │   disclosed to (since): 'Alice' (6), 'Bob' (6), 'USD_Bank' (6)
      └─> 'Bob' exercises Iou_Transfer on #3:1 (Iou:Iou)
            with
               newOwner = 'Alice'
         children:
         #6:4
         │   consumed by: #6:5
         │   referenced by #6:5
         │   disclosed to (since): 'Alice' (6), 'Bob' (6), 'USD_Bank' (6)
         └─> 'Bob' and 'USD_Bank' create Iou:IouTransfer
                                  with
                                     iou =
                                        (Iou:Iou with
                                           issuer = 'USD_Bank';
                                           owner = 'Bob';
                                           currency = "USD";
                                           amount = 110.0000000000;
                                           observers = []);
                                     newOwner = 'Alice'

      #6:5
      │   disclosed to (since): 'Alice' (6), 'Bob' (6), 'USD_Bank' (6)
      └─> 'Alice' exercises IouTransfer_Accept on #6:4 (Iou:IouTransfer)
         children:
         #6:6
         │   disclosed to (since): 'Alice' (6), 'Bob' (6), 'USD_Bank' (6)
         └─> 'Alice' and 'USD_Bank' create Iou:Iou
                                    with
                                       issuer = 'USD_Bank';
```

(continues on next page)

(continued from previous page)

```
                                    owner = 'Alice';
                                    currency = "USD";
                                    amount = 110.0000000000;
                                    observers = []

#6:7
|   disclosed to (since): 'Alice' (6), 'Bob' (6), 'EUR_Bank' (6)
└─> 'Alice' exercises Iou_Transfer on #4:1 (Iou:Iou)
            with
            newOwner = 'Bob'
   children:
   #6:8
   |   consumed by: #6:9
   |   referenced by #6:9
   |   disclosed to (since): 'Alice' (6), 'Bob' (6), 'EUR_Bank' (6)
   └─> 'Alice' and 'EUR_Bank' create Iou:IouTransfer
                              with
                                 iou =
                                 (Iou:Iou with
                                    issuer = 'EUR_Bank';
                                    owner = 'Alice';
                                    currency = "EUR";
                                    amount = 100.0000000000;
                                    observers = ['Bob']);
                                 newOwner = 'Bob'

#6:9
|   disclosed to (since): 'Alice' (6), 'Bob' (6), 'EUR_Bank' (6)
└─> 'Bob' exercises IouTransfer_Accept on #6:8 (Iou:IouTransfer)
   children:
   #6:10
   |   disclosed to (since): 'Alice' (6), 'Bob' (6), 'EUR_Bank' (6)
   └─> 'Bob' and 'EUR_Bank' create Iou:Iou
                            with
                               issuer = 'EUR_Bank';
                               owner = 'Bob';
                               currency = "EUR";
                               amount = 100.0000000000;
                               observers = []
```

The `submit` function used in this script tries to perform a transaction and fails if any of the ledger integrity rules are violated. There is also a `submitMustFail` function, which checks that certain transactions are not possible. This is used in `daml/Tests/Iou.daml`, for example, to confirm that the ledger model prevents double spends.

Integrate With the Ledger

A distributed ledger only forms the core of a full Daml application.

To build automations and integrations around the ledger, Daml has *language bindings* for the Ledger API in several programming languages.

To compile the Java integration for the quickstart application, we first need to run the Java codegen on the DAR we built before:

```
daml codegen java
```

Once the code has been generated (into `target/generated-sources` per the instructions in `daml.yaml`), we can compile it using:

```
mvn compile
```

Now, start the Java integration with:

```
mvn exec:java@run-quickstart -Dparty=$(cat output.json | sed 's/\[\"//' | sed 's/
↪".*//')
```

Note that this step requires that the sandbox started *earlier* is still running. If it is not, you'll have to run the `daml sandbox` and `daml script` commands again to get an `output.json` in sync with the new state of the sandbox (party names can change with each sandbox restart).

The application provides REST services on port `8080` to perform basic operations on behalf on `Alice`. For example, check that:

```
curl http://localhost:8080/iou
```

returns, for a newly-created sandbox (where you have just run the init script to get the `output.json` file), something like:

```
{"0":{"issuer":"EUR_Bank::NAMESPACE","owner":"Alice::NAMESPACE","currency":"EUR",
↪"amount":100.0000000000,"observers":[]}}
```

If you still have the same sandbox running against which you have run the Navigator steps above, the output might look more like:

```
{"0":{"issuer":"Alice::NAMESPACE","owner":"Bob::NAMESPACE","currency":"AliceCoin",
↪"amount":1.0000000000,"observers":[]},"1":{"issuer":"USD_Bank::NAMESPACE","owner
↪":"Alice::NAMESPACE","currency":"USD","amount":110.0000000000,"observers":[]}}
```

To start the same application on another port, use the command-line parameter `-Drestport=PORT`. To start it for another party, use `-Dparty=PARTY`. For example, to start the application for Bob on `8081`, run:

```
mvn exec:java@run-quickstart -Drestport=8081 -Dparty=Bob$(cat output.json | sed
↪'s/\[\"//' | sed 's/".*//')
```

The following REST services are included:

> `GET` on `http://localhost:8080/iou` lists all active Ious, and their Ids.
> Note that the Ids exposed by the REST API are not the ledger contract Ids, but integers. You can open the address in your browser or run `curl -X GET http://localhost:8080/iou`.

`GET` on `http://localhost:8080/iou/ID` returns the Iou with Id `ID`.
For example, to get the content of the Iou with Id 0, run:
`curl -X GET http://localhost:8080/iou/0`
`PUT` on `http://localhost:8080/iou` creates a new Iou on the ledger.
To create another *AliceCoin*, run:

```
curl -X PUT -d '{"issuer":"Alice::NAMESPACE","owner":"Alice::NAMESPACE",
↪"currency":"AliceCoin","amount":1.0,"observers":[]}' http://localhost:8080/
↪iou
```

Note that you have to replace `NAMESPACE` with the real namespace assigned by the sandbox; you can find it in `output.json`:

```
ns=$(cat output.json | sed 's/\["Alice::' | sed 's/".*//'); curl -X PUT -d
↪"$(printf '{"issuer":"Alice::%s","owner":"Alice::%s","currency":"AliceCoin",
↪"amount":1.0,"observers":[]}' $ns $ns)" http://localhost:8080/iou
```

`POST` on `http://localhost:8080/iou/ID/transfer` transfers the Iou with Id `ID`.
Check the index of your new *AliceCoin* by listing all active Ious. If you have just run the init script, it will be `0`; if you have run the Navigator section, it will likely be `2`. Once you have the index, you can run:

```
ns=$(cat output.json | sed 's/\["Alice::' | sed 's/".*//'); curl -X POST -
↪d "{\"newOwner\":\"Bob::${ns}\"}" http://localhost:8080/iou/0/transfer
```

to transfer it to Bob. If it's not `0`, just replace the `0` in `iou/0` in the above command.

The automation is based on the *Java bindings* and the output of the *Java code generator*, which are included as a Maven dependency and Maven plugin respectively in the `pom.xml` file created by the template:

```
        <dependency>
            <groupId>com.daml</groupId>
            <artifactId>bindings-rxjava</artifactId>
            <version>__VERSION__</version>
            <exclusions>
                <exclusion>
                    <groupId>com.google.protobuf</groupId>
                    <artifactId>protobuf-lite</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
```

It consists of the application in file `IouMain.java`. It uses the class `Iou` from `Iou.java`, which is generated from the Daml model with the Java code generator. The `Iou` class provides better serialization and de-serialization to JSON via gson. Looking at `src/main/java/com/daml/quickstart/iou/IouMain.java`:

1. A connection to the ledger is established using a `DamlLedgerClient` object.

```
DamlLedgerClient client = DamlLedgerClient.newBuilder(ledgerhost, ledgerport).
↪build();

// Connects to the ledger and runs initial validation.
client.connect();
```

2. An in-memory contract store is initialized. This is intended to provide a live view of all active contracts, with mappings between ledger and external Ids.

```
ConcurrentHashMap<Long, Iou> contracts = new ConcurrentHashMap<>();
BiMap<Long, Iou.ContractId> idMap = Maps.synchronizedBiMap(HashBiMap.
↪create());
AtomicReference<LedgerOffset> acsOffset =
    new AtomicReference<>(LedgerOffset.LedgerBegin.getInstance());
```

3. The Active Contracts Service (ACS) is used to quickly build up the contract store to a recent state.

```
client
    .getActiveContractSetClient()
    .getActiveContracts(Iou.contractFilter(), Collections.singleton(party),
↪true)
    .blockingForEach(
        response -> {
          response.offset.ifPresent(offset -> acsOffset.set(new LedgerOffset.
↪Absolute(offset)));
          response.activeContracts.forEach(
              contract -> {
                long id = idCounter.getAndIncrement();
                contracts.put(id, contract.data);
                idMap.put(id, contract.id);
              });
        });
```

`blockingForEach` is used to ensure that the contract store is consistent with the ledger state at the latest offset observed by the client.

4. The Transaction Service is wired up to update the contract store on occurrences of `ArchiveEvent` and `CreateEvent` for Ious. Since `getTransactions` is called without end offset, it will stream transactions indefinitely, until the application is terminated.

```
    client
        .getTransactionsClient()
        .getTransactions(
            Iou.contractFilter(), acsOffset.get(), Collections.
↪singleton(party), true)
        .forEach(
            t -> {
              for (Event event : t.getEvents()) {
                if (event instanceof CreatedEvent) {
                  CreatedEvent createdEvent = (CreatedEvent) event;
                  long id = idCounter.getAndIncrement();
                  Iou.Contract contract = Iou.Contract.
↪fromCreatedEvent(createdEvent);
                  contracts.put(id, contract.data);
                  idMap.put(id, contract.id);
                } else if (event instanceof ArchivedEvent) {
                  ArchivedEvent archivedEvent = (ArchivedEvent) event;
                  long id =
                      idMap.inverse().get(new Iou.ContractId(archivedEvent.
↪getContractId()));
                  contracts.remove(id);
                  idMap.remove(id);
                }
              }
            });
```

5. Commands are submitted via the Command Submission Service.

```
  var params =
      CommandsSubmission.create(APP_ID, randomUUID().toString(), update.
↪commands())
          .withActAs(party);

  return client.getCommandClient().submitAndWaitForResult(params, update).
↪blockingGet();
```

You can find examples of `Update` instantiations for creating contract and exercising a choice in the bodies of the `transfer` and `iou` endpoints, respectively.

Listing 22: Exercise a choice

```
Map m = g.fromJson(req.body(), Map.class);
Iou.ContractId contractId = idMap.get(Long.parseLong(req.params("id")));
var update = contractId.exerciseIou_Transfer(m.get("newOwner").toString());
```

Listing 23: Create a contract

```
Iou iou = g.fromJson(req.body(), Iou.class);
var iouCreate = iou.create();
var createdContractId = submit(client, party, iouCreate);
```

The rest of the application sets up the REST services using Spark Java, and does dynamic package Id detection using the Package Service. The latter is useful during development when package Ids change frequently.

For a discussion of ledger application design and architecture, take a look at *Application Architecture Guide*.

Next Steps

Great - you've completed the quickstart guide!

Some steps you could take next include:

Explore examples for guidance and inspiration.
Learn Daml.
Language reference.
Learn more about *application development*.
Learn about the *conceptual models* behind Daml.

1.12.5.6 Python Bindings

The Python bindings (formerly known as DAZL) are a client implementation of the *Ledger API* for the Python language and are supported under the Daml Enterprise license.

The Python bindings are supported for use with Daml and with Daml Hub. Documentation for the bindings can be found here.

1.12.5.7 Use the Ledger API With gRPC

If you want to write an application for the ledger API in other languages, you'll need to use gRPC directly.

If you're not familiar with gRPC and protobuf, we strongly recommend following the gRPC quickstart and gRPC tutorials. This documentation is written assuming you already have an understanding of gRPC.

Get Started

You can get the protobufs from a GitHub release, or from the `daml` repository here.

Protobuf Reference Documentation

For full details of all of the Ledger API services and their RPC methods, see *Ledger API Reference*.

Example Project

We have an example project demonstrating the use of the Ledger API with gRPC. To get the example project, `PingPongGrpc`:

1. Configure your machine to use the example by following the instructions at *Set Up a Maven Project*.
2. Clone the repository from GitHub.
3. Follow the setup instructions in the README. Use `examples.pingpong.grpc.PingPongGrpcMain` as the main class.

About the Example Project

The example shows very simply how two parties can interact via a ledger, using two Daml contract templates, `Ping` and `Pong`.

The logic of the application goes like this:

1. The application injects a contract of type `Ping` for `Alice`.
2. `Alice` sees this contract and exercises the consuming choice `RespondPong` to create a contract of type `Pong` for `Bob`.
3. `Bob` sees this contract and exercises the consuming choice `RespondPing` to create a contract of type `Ping` for `Alice`.
4. Points 2 and 3 are repeated until the maximum number of contracts defined in the Daml is reached.

The entry point for the Java code is the main class `src/main/java/examples/pingpong/grpc/PingPongGrpcMain.java`. Look at it to see how connect to and interact with a ledger using gRPC.

The application prints output like this:

```
Bob is exercising RespondPong on #1:0 in workflow Ping-Alice-1 at count 0
Alice is exercising RespondPing on #344:1 in workflow Ping-Alice-7 at count 9
```

The first line shows:

> `Bob` is exercising the `RespondPong` choice on the contract with ID `#1:0` for the workflow `Ping-Alice-1`.
> Count `0` means that this is the first choice after the initial `Ping` contract.
> The workflow ID `Ping-Alice-1` conveys that this is the workflow triggered by the second initial `Ping` contract that was created by `Alice`.

This example subscribes to transactions for a single party, as different parties typically live on different participant nodes. However, if you have multiple parties registered on the same node, or are running an application against the Sandbox, you can subscribe to transactions for multiple parties in a single subscription by putting multiple entries into the `filters_by_party` field of the `TransactionFilter` message. Subscribing to transactions for an unknown party will result in an error.

Daml Types and Protobuf

For information on how Daml types and contracts are represented by the Ledger API as protobuf messages, see *How Daml Types are Translated to Protobuf*.

Error Handling

The Ledger API generally uses the gRPC standard status codes for signaling response failures to client applications.

For more details on the gRPC standard status codes, see the gRPC documentation .

Generically, on submitted commands the Ledger API responds with the following gRPC status codes:

ABORTED The platform failed to record the result of the command due to a transient server-side error (e.g. backpressure due to high load) or a time constraint violation. You can retry the submission. In case of a time constraint violation, please refer to the section *Dealing with time* on how to handle commands with long processing times.

DEADLINE_EXCEEDED (when returned by the Command Service) The request might not have been processed, as its deadline expired before its completion was signalled.

ALREADY_EXISTS The command was rejected because the resource (e.g. contract key) already exists or because it was sent within the deduplication period of a previous command with the same change ID.

NOT_FOUND The command was rejected due to a missing resources (e.g. contract key not found).

INVALID_ARGUMENT The submission failed because of a client error. The platform will definitely reject resubmissions of the same command.

FAILED_PRECONDITION The command was rejected due to an interpretation error or due to a consistency error due to races.

OK (when returned by the Command Submission Service) Assume that the command was accepted and wait for the resulting completion or a timeout from the Command Completion Service.

OK (when returned by the Command Service) You can be sure that the command was successful.

INTERNAL, UNKNOWN (when returned by the Command Service) An internal system fault occurred. Contact the participant operator for the resolution.

Aside from the standard gRPC status codes, the failures returned by the Ledger API are enriched with details meant to help the application or the application developer to handle the error autonomously (e.g. by retrying on a retryable error). For more details on the rich error details see the *Error Codes*

1.12.5.8 Ledger API Reference

com/daml/ledger/api/v1/active_contracts_service.proto

GetActiveContractsRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as described in `value.proto`). Optional
filter	*TransactionFilter*		Templates to include in the served snapshot, per party. Required
verbose	*bool*		If enabled, values served over the API will contain more information than strictly necessary to interpret the data. In particular, setting the verbose flag to true triggers the ledger to include labels for record fields. Optional
active_at_offset	*string*		The offset at which the snapshot of the active contracts will be computed. Must be no greater than the current ledger end offset. Must be greater than or equal to the last pruning offset. If not set the current ledger end offset will be used. Optional

GetActiveContractsResponse

Field	Type	Label	Description
offset	*string*		Included only in the last message. The client should start consuming the transactions endpoint with this offset. The format of this field is described in `ledger_offset.proto`.
workflow_id	*string*		The workflow that created the contracts. Must be a valid LedgerString (as described in `value.proto`).
active_contracts	*CreatedEvent*	repeated	The list of contracts that were introduced by the workflow with `workflow_id` at the offset. Must be a valid LedgerString (as described in `value.proto`).

ActiveContractsService

Allows clients to initialize themselves according to a fairly recent state of the ledger without reading through all transactions that were committed since the ledger's creation. In V2 Ledger API this service is not available anymore. Use v2.StateService instead.

Method name	Request type	Response type	Description
GetActiveContracts	*GetActiveContractsRequest*	*GetActiveContractsResponse*	Returns a stream of the snapshot of the active contracts at a ledger offset. If there are no active contracts, the stream returns a single response message with the offset at which the snapshot has been taken. Clients SHOULD use the offset in the last GetActiveContractsResponse message to continue streaming transactions with the transaction service. Clients SHOULD NOT assume that the set of active contracts they receive reflects the state at the ledger end.

com/daml/ledger/api/v1/admin/config_management_service.proto

GetTimeModelRequest

GetTimeModelResponse

Field	Type	Label	Description
configuration_generation	*int64*		The current configuration generation. The generation is a monotonically increasing integer that is incremented on each change. Used when setting the time model.
time_model	*TimeModel*		The current ledger time model.

SetTimeModelRequest

Field	Type	Label	Description
submission_id	*string*		Submission identifier used for tracking the request and to reject duplicate submissions. Required.
maximum_record_time	google.protobuf.Timestamp		Deadline for the configuration change after which the change is rejected.
configuration_generation	*int64*		The current configuration generation which we're submitting the change against. This is used to perform a compare-and-swap of the configuration to safeguard against concurrent modifications. Required.
new_time_model	*TimeModel*		The new time model that replaces the current one. Required.

SetTimeModelResponse

Field	Type	Label	Description
configuration_generation	*int64*		The configuration generation of the committed time model.

TimeModel

Field	Type	Label	Description
avg_transaction_latency	google.protobuf.Duration		The expected average latency of a transaction, i.e., the average time from submitting the transaction to a [[WriteService]] and the transaction being assigned a record time. Required.
min_skew	google.protobuf.Duration		The minimimum skew between ledger time and record time: lt_TX >= rt_TX - minSkew Required.
max_skew	google.protobuf.Duration		The maximum skew between ledger time and record time: lt_TX <= rt_TX + maxSkew Required.

ConfigManagementService

Status: experimental interface, will change before it is deemed production ready

The ledger configuration management service provides methods for the ledger administrator to change the current ledger configuration. The services provides methods to modify different aspects of the configuration. In V2 Ledger API this service is not available anymore.

Method name	Request type	Response type	Description
GetTimeModel	*GetTimeModelRequest*	*GetTimeModelResponse*	Return the currently active time model and the current configuration generation.
SetTimeModel	*SetTimeModelRequest*	*SetTimeModelResponse*	Set the ledger time model.

com/daml/ledger/api/v1/admin/identity_provider_config_service.proto

CreateIdentityProviderConfigRequest

Field	Type	Label	Description
identity_provider_config	*IdentityProviderConfig*		Required

CreateIdentityProviderConfigResponse

Field	Type	Label	Description
identity_provider_config	*IdentityProviderConfig*		

DeleteIdentityProviderConfigRequest

Field	Type	Label	Description
identity_provider_id	*string*		The identity provider config to delete. Required

DeleteIdentityProviderConfigResponse

Does not (yet) contain any data.

GetIdentityProviderConfigRequest

Field	Type	Label	Description
identity_provider_id	*string*		Required

GetIdentityProviderConfigResponse

Field	Type	Label	Description
identity_provider_config	*IdentityProviderConfig*		

IdentityProviderConfig

Field	Type	Label	Description
identity_provider_id	*string*		The identity provider identifier Must be a valid LedgerString (as describe in `value.proto`). Required
is_deactivated	*bool*		When set, the callers using JWT tokens issued by this identity provider are denied all access to the Ledger API. Optional, Modifiable
issuer	*string*		Specifies the issuer of the JWT token. The issuer value is a case sensitive URL using the https scheme that contains scheme, host, and optionally, port number and path components and no query or fragment components. Required Modifiable
jwks_url	*string*		The JWKS (JSON Web Key Set) URL. The Ledger API uses JWKs (JSON Web Keys) from the provided URL to verify that the JWT has been signed with the loaded JWK. Only RS256 (RSA Signature with SHA-256) signing algorithm is supported. Required Modifiable
audience	*string*		Specifies the audience of the JWT token. When set, the callers using JWT tokens issued by this identity provider are allowed to get an access only if the aud claim includes the string specified here Optional, Modifiable

ListIdentityProviderConfigsRequest

Pagination is not required as the resulting data set is small enough to be returned in a single call

ListIdentityProviderConfigsResponse

Field	Type	Label	Description
identity_provider_configs	*IdentityProviderConfig*	repeated	

UpdateIdentityProviderConfigRequest

Field	Type	Label	Description
identity_provider_config	*IdentityProviderConfig*		The identity provider config to update. Required, Modifiable
update_mask	google.protobuf.FieldMask		An update mask specifies how and which properties of the `IdentityProviderConfig` message are to be updated. An update mask consists of a set of update paths. A valid update path points to a field or a subfield relative to the `IdentityProviderConfig` message. A valid update mask must: (1) contain at least one update path, (2) contain only valid update paths. Fields that can be updated are marked as `Modifiable`. For additional information see the documentation for standard protobuf3's `google.protobuf.FieldMask`. Required

UpdateIdentityProviderConfigResponse

Field	Type	Label	Description
identity_provider_config	*IdentityProviderConfig*		Updated identity provider config

IdentityProviderConfigService

Identity Provider Config Service makes it possible for participant node administrators to setup and manage additional identity providers at runtime.

This allows using access tokens from identity providers unknown at deployment time. When an identity provider is configured, independent IDP administrators can manage their own set of parties and users. Such parties and users have a matching *identity_provider_id* defined and are inaccessible to administrators from other identity providers. A user will only be authenticated if the corresponding JWT token is issued by the appropriate identity provider. Users and parties without *identity_provider_id* defined are assumed to be using the default identity provider, which is configured statically at the participant node's deployment time.

The Ledger API uses the `iss` claim of a JWT token to match the token to a specific IDP. If there is no match, the default IDP is assumed.

The fields of request messages (and sub-messages) are marked either as `Optional` or `Required`: (1) `Optional` denoting the client may leave the field unset when sending a request. (2) `Required` denoting the client must set the field to a non-default value when sending a request.

An identity provider config resource is described by the `IdentityProviderConfig` message, An identity provider config resource, once it has been created, can be modified. In order to update the properties represented by the `IdentityProviderConfig` message use the `UpdateIdentityProviderConfig` RPC. The only fields that can be modified are those marked as `Modifiable`.

Method name	Request type	Response type	Description
CreateIdentityProviderConfig	*CreateIdentityProviderConfigRequest*	*CreateIdentityProviderConfigResponse*	Create a new identity provider configuration. The request will fail if the maximum allowed number of separate configurations is reached.
GetIdentityProviderConfig	*GetIdentityProviderConfigRequest*	*GetIdentityProviderConfigResponse*	Get the identity provider configuration data by id.
UpdateIdentityProviderConfig	*UpdateIdentityProviderConfigRequest*	*UpdateIdentityProviderConfigResponse*	Update selected modifiable attribute of an identity provider config resource described by the `IdentityProviderConfig` message.
ListIdentityProviderConfigs	*ListIdentityProviderConfigsRequest*	*ListIdentityProviderConfigsResponse*	List all existing identity provider configurations.
DeleteIdentityProviderConfig	*DeleteIdentityProviderConfigRequest*	*DeleteIdentityProviderConfigResponse*	Delete an existing identity provider configuration.

com/daml/ledger/api/v1/admin/metering_report_service.proto

GetMeteringReportRequest

Authorized if and only if the authenticated user is a participant admin.

Field	Type	Label	Description
from	google.protobuf.Timestamp		The from timestamp (inclusive). Required.
to	google.protobuf.Timestamp		The to timestamp (exclusive). If not provided, the server will default to its current time.
application_id	*string*		If set to a non-empty value, then the report will only be generated for that application. Optional.

GetMeteringReportResponse

Field	Type	Label	Description
request	*GetMeteringReportRequest*		The actual request that was executed.
report_generation_time	google.protobuf.Timestamp		The time at which the report was computed.
metering_report_json	google.protobuf.Struct		The metering report json. For a JSON Schema definition of the JSon see: https://github.com/digital-asset/daml/blob/main/ledger-api/grpc-definitions/metering-report-schema.json

MeteringReportService

Experimental API to retrieve metering reports.

Metering reports aim to provide the information necessary for billing participant and application operators.

Method name	Request type	Response type	Description
GetMeteringReport	*GetMeteringReportRequest*	*GetMeteringReportResponse*	Retrieve a metering report.

com/daml/ledger/api/v1/admin/object_meta.proto

ObjectMeta

Represents metadata corresponding to a participant resource (e.g. a participant user or participant local information about a party).

Based on `ObjectMeta` meta used in Kubernetes API. See https://github.com/kubernetes/apimachinery/blob/master/pkg/apis/meta/v1/generated.proto#L640

Field	Type	Label	Description
resource_version	*string*		An opaque, non-empty value, populated by a participant server which represents the internal version of the resource this `ObjectMeta` message is attached to. The participant server will change it to a unique value each time the corresponding resource is updated. You must not rely on the format of resource version. The participant server might change it without notice. You can obtain the newest resource version value by issuing a read request. You may use it for concurrent change detection by passing it back unmodified in an update request. The participant server will then compare the passed value with the value maintained by the system to determine if any other updates took place since you had read the resource version. Upon a successful update you are guaranteed that no other update took place during your read-modify-write sequence. However, if another update took place during your read-modify-write sequence then your update will fail with an appropriate error. Concurrent change control is optional. It will be applied only if you include a resource version in an update request. When creating a new instance of a resource you must leave the resource version empty. Its value will be populated by the participant server upon successful resource creation. Optional
annotations	*ObjectMeta.AnnotationsEntry*	repeated	A set of modifiable key-value pairs that can be used to represent arbitrary, client-specific metadata. Constraints: 1. The total size over all keys and values cannot exceed 256kb in UTF-8 encoding. 2. Keys are composed of an optional prefix segment and a required name segment such that: - key prefix, when present, must be a valid DNS subdomain with at most 253 characters, followed by a '/' (forward slash) character, - name segment must have at most 63 characters that are either alphanumeric ([a-z0-9A-Z]), or a '.' (dot), '-' (dash) or '_' (underscore); and it must start and end with an alphanumeric character. 2. Values can be any non-empty strings. Keys with empty prefix are reserved for end-users. Properties set by external tools or internally by the participant server must use non-empty key prefixes. Duplicate keys are disallowed by the semantics of the protobuf3 maps. See: https://developers.google.com/protocol-buffers/docs/proto3#maps Annotations may be a part of a modifiable resource. Use the resource's update RPC to update its annotations. In order to add a new annotation or update an existing one using an update RPC, provide the desired annotation in the update request. In order to remove an annotation using an update RPC, provide the target annotation's key but set its value to the empty string in the update request. Optional Modifiable

ObjectMeta.AnnotationsEntry

Field	Type	Label	Description
key	*string*		
value	*string*		

com/daml/ledger/api/v1/admin/package_management_service.proto

ListKnownPackagesRequest

ListKnownPackagesResponse

Field	Type	Label	Description
package_details	*PackageDetails*	repeated	The details of all Daml-LF packages known to backing participant. Required

PackageDetails

Field	Type	Label	Description
package_id	*string*		The identity of the Daml-LF package. Must be a valid PackageIdString (as describe in `value.proto`). Required
package_size	*uint64*		Size of the package in bytes. The size of the package is given by the size of the `daml_lf` ArchivePayload. See further details in `daml_lf.proto`. Required
known_since	google.protobuf.Timestamp		Indicates since when the package is known to the backing participant. Required
source_description	*string*		Description provided by the backing participant describing where it got the package from. Optional

UploadDarFileRequest

Field	Type	Label	Description
dar_file	*bytes*		Contains a Daml archive DAR file, which in turn is a jar like zipped container for `daml_lf` archives. See further details in `daml_lf.proto`. Required
submission_id	*string*		Unique submission identifier. Optional, defaults to a random identifier.

UploadDarFileResponse

An empty message that is received when the upload operation succeeded.

PackageManagementService

Status: experimental interface, will change before it is deemed production ready

Query the Daml-LF packages supported by the ledger participant and upload DAR files. We use 'backing participant' to refer to this specific participant in the methods of this API.

Method name	Request type	Response type	Description
ListKnownPackages	*ListKnownPackagesRequest*	*ListKnownPackagesResponse*	Returns the details of all Daml-LF packages known to the backing participant.
UploadDarFile	*UploadDarFileRequest*	*UploadDarFileResponse*	Upload a DAR file to the backing participant. Depending on the ledger implementation this might also make the package available on the whole ledger. This call might not be supported by some ledger implementations. Canton could be an example, where uploading a DAR is not sufficient to render it usable, it must be activated first. This call may: - Succeed, if the package was successfully uploaded, or if the same package was already uploaded before. - Respond with a gRPC error

com/daml/ledger/api/v1/admin/participant_pruning_service.proto

PruneRequest

Field	Type	Label	Description
prune_up_to	*string*		Inclusive offset up to which the ledger is to be pruned. By default the following data is pruned: 1. All normal and divulged contracts that have been archived before *prune_up_to*. 2. All transaction events and completions before *prune_up_to*
submission_id	*string*		Unique submission identifier. Optional, defaults to a random identifier, used for logging.
prune_all_divulged_contracts	*bool*		Prune all immediately and retroactively divulged contracts created before *prune_up_to* independent of whether they were archived before *prune_up_to*. Useful to avoid leaking storage on participant nodes that can see a divulged contract but not its archival.

Application developers SHOULD write their Daml applications such that they do not rely on divulged contracts; i.e., no warnings from using divulged contracts as inputs to transactions are emitted.

Participant node operators SHOULD set the *prune_all_divulged_contracts* flag to avoid leaking storage due to accumulating unarchived divulged contracts PROVIDED that: 1. no application using this

participant node relies on divulgence OR 2. divulged contracts on which applications rely have been re-divulged after the *prune_up_to* offset.

PruneResponse

Empty for now, but may contain fields in the future

ParticipantPruningService

Prunes/truncates the oldest transactions from the participant (the participant Ledger Api Server plus any other participant-local state) by removing a portion of the ledger in such a way that the set of future, allowed commands are not affected.

This enables: 1. keeping the inactive portion of the ledger to a manageable size and 2. removing inactive state to honor the right to be forgotten.

Method name	Request type	Response type	Description
Prune	*PruneRequest*	*PruneResponse*	Prune the ledger specifying the offset before and at which ledger transactions should be removed. Only returns when the potentially long-running prune request ends successfully or with an error.

com/daml/ledger/api/v1/admin/party_management_service.proto

AllocatePartyRequest

Required authorization: `HasRight(ParticipantAdmin) OR IsAuthenticatedIdentityProviderAdmin(identity_provider_id)`

Field	Type	Label	Description
party_id_hint	*string*		A hint to the participant which party ID to allocate. It can be ignored. Must be a valid PartyIdString (as described in `value.proto`). Optional
display_name	*string*		Human-readable name of the party to be added to the participant. It doesn't have to be unique. Use of this field is discouraged. Use `local_metadata` instead. Optional
local_metadata	*ObjectMeta*		Participant-local metadata to be stored in the `PartyDetails` of this newly allocated party. Optional
identity_provider_id	*string*		The id of the `Identity Provider` Optional, if not set, assume the party is managed by the default identity provider or party is not hosted by the participant.

AllocatePartyResponse

Field	Type	Label	Description
party_details	*PartyDetails*		

GetParticipantIdRequest

Required authorization: `HasRight(ParticipantAdmin)`

GetParticipantIdResponse

Field	Type	Label	Description
participant_id	*string*		Identifier of the participant, which SHOULD be globally unique. Must be a valid LedgerString (as describe in `value.proto`).

GetPartiesRequest

Required authorization: `HasRight(ParticipantAdmin) OR IsAuthenticatedIdentityProviderAdmin(identity_provider_id)`

Field	Type	Label	Description
parties	*string*	repeated	The stable, unique identifier of the Daml parties. Must be valid PartyIdStrings (as described in `value.proto`). Required
identity_provider_id	*string*		The id of the `Identity Provider` whose parties should be retrieved. Optional, if not set, assume the party is managed by the default identity provider or party is not hosted by the participant.

GetPartiesResponse

Field	Type	Label	Description
party_details	*PartyDetails*	repeated	The details of the requested Daml parties by the participant, if known. The party details may not be in the same order as requested. Required

ListKnownPartiesRequest

Required authorization: `HasRight(ParticipantAdmin) OR IsAuthenticatedIdentityProviderAdmin(identity_provider_id)`

Field	Type	Label	Description
identity_provider_id	*string*		The id of the `Identity Provider` whose parties should be retrieved. Optional, if not set, assume the party is managed by the default identity provider or party is not hosted by the participant.

ListKnownPartiesResponse

Field	Type	Label	Description
party_details	*PartyDetails*	repeated	The details of all Daml parties known by the participant. Required

PartyDetails

Field	Type	Label	Description
party	*string*		The stable unique identifier of a Daml party. Must be a valid PartyIdString (as described in `value.proto`). Required
display_name	*string*		Human readable name associated with the party at allocation time. Caution, it might not be unique. Use of this field is discouraged. Use the *local_metadata* field instead. Optional
is_local	*bool*		true if party is hosted by the participant and the party shares the same identity provider as the user issuing the request. Optional
local_metadata	*ObjectMeta*		Participant-local metadata of this party. Optional, Modifiable
identity_provider_id	*string*		The id of the `Identity Provider` Optional, if not set, there could be 3 options: 1) the party is managed by the default identity provider. 2) party is not hosted by the participant. 3) party is hosted by the participant, but is outside of the user's identity provider.

UpdatePartyDetailsRequest

Required authorization: `HasRight(ParticipantAdmin) OR IsAuthenticatedIdentityProviderAdmi identity_provider_id)`

Field	Type	Label	Description
party_details	*PartyDetails*		Party to be updated Required, Modifiable
update_mask	google.protobuf.FieldMask		An update mask specifies how and which properties of the `PartyDetails` message are to be updated. An update mask consists of a set of update paths. A valid update path points to a field or a subfield relative to the `PartyDetails` message. A valid update mask must: (1) contain at least one update path, (2) contain only valid update paths. Fields that can be updated are marked as `Modifiable`. An update path can also point to non-`Modifiable` fields such as 'party' and 'local_metadata.resource_version' because they are used: (1) to identify the party details resource subject to the update, (2) for concurrent change control. An update path can also point to non-`Modifiable` fields such as 'is_local' and 'display_name' as long as the values provided in the update request match the server values. Examples of update paths: 'local_metadata.annotations', 'local_metadata'. For additional information see the documentation for standard protobuf3's `google.protobuf.FieldMask`. For similar Ledger API see `com.daml.ledger.api.v1.admin.UpdateUserRequest`. Required

UpdatePartyDetailsResponse

Field	Type	Label	Description
party_details	*PartyDetails*		Updated party details

UpdatePartyIdentityProviderRequest

Required authorization: `HasRight(ParticipantAdmin)`

Field	Type	Label	Description
party	*string*		Party to update
source_identity_provider_id	*string*		Current identity provider id of the party
target_identity_provider_id	*string*		Target identity provider id of the party

UpdatePartyIdentityProviderResponse

PartyManagementService

This service allows inspecting the party management state of the ledger known to the participant and managing the participant-local party metadata.

The authorization rules for its RPCs are specified on the `<RpcName>Request` messages as boolean expressions over these facts: (1) `HasRight(r)` denoting whether the authenticated user has right `r` and (2) `IsAuthenticatedIdentityProviderAdmin(idp)` denoting whether `idp` is equal to the `identity_provider_id` of the authenticated user and the user has an IdentityProviderAdmin right. If *identity_provider_id* is set to an empty string, then it's effectively set to the value of access token's 'iss' field if that is provided. If *identity_provider_id* remains an empty string, the default identity provider will be assumed.

The fields of request messages (and sub-messages) are marked either as `Optional` or `Required`: (1) `Optional` denoting the client may leave the field unset when sending a request. (2) `Required` denoting the client must set the field to a non-default value when sending a request.

A party details resource is described by the `PartyDetails` message, A party details resource, once it has been created, can be modified using the `UpdatePartyDetails` RPC. The only fields that can be modified are those marked as `Modifiable`.

Method name	Request type	Response type	Description
GetParticipantId	*GetParticipantIdRequest*	*GetParticipantIdResponse*	Return the identifier of the participant. All horizontally scaled replicas should return the same id. daml-on-kv-ledger: returns an identifier supplied on command line at launch time canton: returns globally unique identifier of the participant
GetParties	*GetPartiesRequest*	*GetPartiesResponse*	Get the party details of the given parties. Only known parties will be returned in the list.
ListKnownParties	*ListKnownPartiesRequest*	*ListKnownPartiesResponse*	List the parties known by the participant. The list returned contains parties whose ledger access is facilitated by the participant and the ones maintained elsewhere.
AllocateParty	*AllocatePartyRequest*	*AllocatePartyResponse*	Allocates a new party on a ledger and adds it to the set managed by the participant. Caller specifies a party identifier suggestion, the actual identifier allocated might be different and is implementation specific. Caller can specify party metadata that is stored locally on the participant. This call may: - Succeed, in which case the actual allocated identifier is visible in the response. - Respond with a gRPC error daml-on-kv-ledger: suggestion's uniqueness is checked by the validators in the consensus layer and call rejected if the identifier is already present. canton: completely different globally unique identifier is allocated. Behind the scenes calls to an internal protocol are made. As that protocol is richer than the surface protocol, the arguments take implicit values The party identifier suggestion must be a valid party name. Party names are required to be non-empty US-ASCII strings built from letters, digits, space, colon, minus and underscore limited to 255 chars
UpdatePartyDetails	*UpdatePartyDetailsRequest*	*UpdatePartyDetailsResponse*	Update selected modifiable participant-local attributes of a party details resource. Can update the participant's local information for local parties.
UpdatePartyIdentityProviderId	*UpdatePartyIdentityProviderRequest*	*UpdatePartyIdentityProviderResponse*	Update the assignment of a party from one IDP to another.

com/daml/ledger/api/v1/admin/user_management_service.proto

CreateUserRequest

Required authorization: `HasRight(ParticipantAdmin) OR IsAuthenticatedIdentityProviderAdmi identity_provider_id)`

Field	Type	Label	Description
user	*User*		The user to create. Required
rights	*Right*	repeated	The rights to be assigned to the user upon creation, which SHOULD include appropriate rights for the `user.primary_party`. Optional

CreateUserResponse

Field	Type	Label	Description
user	*User*		Created user.

DeleteUserRequest

Required authorization: `HasRight(ParticipantAdmin) OR IsAuthenticatedIdentityProviderAdmin(identity_provider_id)`

Field	Type	Label	Description
user_id	*string*		The user to delete. Required
identity_provider_id	*string*		The id of the `Identity Provider` Optional, if not set, assume the user is managed by the default identity provider.

DeleteUserResponse

Does not (yet) contain any data.

GetUserRequest

Required authorization: `HasRight(ParticipantAdmin) OR IsAuthenticatedIdentityProviderAdmin(identity_provider_id) OR IsAuthenticatedUser(user_id)`

Field	Type	Label	Description
user_id	*string*		The user whose data to retrieve. If set to empty string (the default), then the data for the authenticated user will be retrieved. Optional
identity_provider_id	*string*		The id of the `Identity Provider` Optional, if not set, assume the user is managed by the default identity provider.

GetUserResponse

Field	Type	Label	Description
user	*User*		Retrieved user.

GrantUserRightsRequest

Add the rights to the set of rights granted to the user.

Required authorization: `HasRight(ParticipantAdmin) OR IsAuthenticatedIdentityProviderAdmin(identity_provider_id)`

Field	Type	Label	Description
user_id	*string*		The user to whom to grant rights. Required
rights	*Right*	repeated	The rights to grant. Optional
identity_provider_id	*string*		The id of the `Identity Provider` Optional, if not set, assume the user is managed by the default identity provider.

GrantUserRightsResponse

Field	Type	Label	Description
newly_granted_rights	*Right*	repeated	The rights that were newly granted by the request.

ListUserRightsRequest

Required authorization: `HasRight(ParticipantAdmin) OR IsAuthenticatedIdentityProviderAdmin(identity_provider_id) OR IsAuthenticatedUser(user_id)`

Field	Type	Label	Description
user_id	*string*		The user for which to list the rights. If set to empty string (the default), then the rights for the authenticated user will be listed. Required
identity_provider_id	*string*		The id of the `Identity Provider` Optional, if not set, assume the user is managed by the default identity provider.

ListUserRightsResponse

Field	Type	Label	Description
rights	*Right*	repeated	All rights of the user.

ListUsersRequest

Required authorization: `HasRight(ParticipantAdmin) OR IsAuthenticatedIdentityProviderAdmin(identity_provider_id)`

Field	Type	Label	Description
page_token	*string*		Pagination token to determine the specific page to fetch. Leave empty to fetch the first page. Optional
page_size	*int32*		Maximum number of results to be returned by the server. The server will return no more than that many results, but it might return fewer. If 0, the server will decide the number of results to be returned. Optional
identity_provider_id	*string*		The id of the `Identity Provider` Optional, if not set, assume the user is managed by the default identity provider.

ListUsersResponse

Field	Type	Label	Description
users	*User*	repeated	A subset of users of the participant node that fit into this page.
next_page_token	*string*		Pagination token to retrieve the next page. Empty, if there are no further results.

RevokeUserRightsRequest

Remove the rights from the set of rights granted to the user.

Required authorization: `HasRight(ParticipantAdmin) OR IsAuthenticatedIdentityProviderAdmin(identity_provider_id)`

Field	Type	Label	Description
user_id	*string*		The user from whom to revoke rights. Required
rights	*Right*	repeated	The rights to revoke. Optional
identity_provider_id	*string*		The id of the `Identity Provider` Optional, if not set, assume the user is managed by the default identity provider.

RevokeUserRightsResponse

Field	Type	Label	Description
newly_revoked_rights	*Right*	repeated	The rights that were actually revoked by the request.

Right

A right granted to a user.

Field	Type	Label	Description
oneof kind.participant_admin	*Right.ParticipantAdmin*		The user can administer the participant node.
oneof kind.can_act_as	*Right.CanActAs*		The user can act as a specific party.
oneof kind.can_read_as	*Right.CanReadAs*		The user can read ledger data visible to a specific party.
oneof kind.identity_provider_admin	*Right.IdentityProviderAdmin*		The user can administer users and parties assigned to the same identity provider as the one of the user.

Right.CanActAs

Field	Type	Label	Description
party	*string*		The right to authorize commands for this party.

Right.CanReadAs

Field	Type	Label	Description
party	*string*		The right to read ledger data visible to this party.

Right.IdentityProviderAdmin

The right to administer the identity provider that the user is assigned to. It means, being able to manage users and parties that are also assigned to the same identity provider.

Right.ParticipantAdmin

The right to administer the participant node.

UpdateUserIdentityProviderRequest

Required authorization: `HasRight(ParticipantAdmin)`

Field	Type	Label	Description
user_id	*string*		User to update
source_identity_provider_id	*string*		Current identity provider id of the user
target_identity_provider_id	*string*		Target identity provider id of the user

UpdateUserIdentityProviderResponse

UpdateUserRequest

Required authorization: `HasRight(ParticipantAdmin) OR IsAuthenticatedIdentityProviderAdmi identity_provider_id)`

Field	Type	Label	Description
user	*User*		The user to update. Required, Modifiable
update_mask	google.protobuf.FieldMask		An update mask specifies how and which properties of the `User` message are to be updated. An update mask consists of a set of update paths. A valid update path points to a field or a subfield relative to the `User` message. A valid update mask must: (1) contain at least one update path, (2) contain only valid update paths. Fields that can be updated are marked as `Modifiable`. An update path can also point to a non-`Modifiable` fields such as 'id' and 'metadata.resource_version' because they are used: (1) to identify the user resource subject to the update, (2) for concurrent change control. Examples of valid update paths: 'primary_party', 'metadata', 'metadata.annotations'. For additional information see the documentation for standard protobuf3's `google.protobuf.FieldMask`. For similar Ledger API see `com.daml.ledger.api.v1.admin.UpdatePartyDetailsRequest`. Required

UpdateUserResponse

Field	Type	Label	Description
user	*User*		Updated user

User

Users are used to dynamically manage the rights given to Daml applications. They are stored and managed per participant node.

Read the *Authorization documentation* to learn more.

Field	Type	Label	Description
id	*string*		The user identifier, which must be a non-empty string of at most 128 characters that are either alphanumeric ASCII characters or one of the symbols @^$.!`-#+'~_|: . Required
primary_party	*string*		The primary party as which this user reads and acts by default on the ledger *provided* it has the corresponding `CanReadAs(primary_party)` or `CanActAs(primary_party)` rights. Ledger API clients SHOULD set this field to a non-empty value for all users to enable the users to act on the ledger using their own Daml party. Users for participant administrators MAY have an associated primary party. Optional, Modifiable
is_deactivated	*bool*		When set, then the user is denied all access to the Ledger API. Otherwise, the user has access to the Ledger API as per the user's rights. Optional, Modifiable
metadata	*ObjectMeta*		The metadata of this user. Note that the `metadata.resource_version` tracks changes to the properties described by the `User` message and not the user's rights. Optional, Modifiable
identity_provider_id	*string*		The id of the identity provider configured by `Identity Provider Config` Optional, if not set, assume the user is managed by the default identity provider.

UserManagementService

Service to manage users and their rights for interacting with the Ledger API served by a participant node.

The authorization rules for its RPCs are specified on the `<RpcName>Request` messages as boolean expressions over these facts: (1) `HasRight(r)` denoting whether the authenticated user has right `r` and (2) `IsAuthenticatedUser(uid)` denoting whether `uid` is the empty string or equal to the id of the authenticated user. (3) `IsAuthenticatedIdentityProviderAdmin(idp)` denoting whether `idp` is equal to the `identity_provider_id` of the authenticated user and the user has an IdentityProviderAdmin right. If *user_id* is set to the empty string (the default), then the data for the authenticated user will be retrieved. If *identity_provider_id* is set to an empty string, then it's effectively set

to the value of access token's 'iss' field if that is provided. If *identity_provider_id* remains an empty string, the default identity provider will be assumed.

The fields of request messages (and sub-messages) are marked either as `Optional` or `Required`: (1) `Optional` denoting the client may leave the field unset when sending a request. (2) `Required` denoting the client must set the field to a non-default value when sending a request.

A user resource consists of: (1) a set of properties represented by the `User` message, (2) a set of user rights, where each right is represented by the `Right` message.

A user resource, once it has been created, can be modified. In order to update the properties represented by the `User` message use the `UpdateUser` RPC. The only fields that can be modified are those marked as `Modifiable`. In order to grant or revoke user rights use `GrantRights' and ``RevokeRights` RPCs.

Method name	Request type	Response type	Description
CreateUser	*CreateUserRequest*	*CreateUserResponse*	Create a new user.
GetUser	*GetUserRequest*	*GetUserResponse*	Get the user data of a specific user or the authenticated user.
UpdateUser	*UpdateUserRequest*	*UpdateUserResponse*	Update selected modifiable attribute of a user resource described by the `User` message.
DeleteUser	*DeleteUserRequest*	*DeleteUserResponse*	Delete an existing user and all its rights.
ListUsers	*ListUsersRequest*	*ListUsersResponse*	List all existing users.
GrantUserRights	*GrantUserRightsRequest*	*GrantUserRightsResponse*	Grant rights to a user. Granting rights does not affect the resource version of the corresponding user.
RevokeUserRights	*RevokeUserRightsRequest*	*RevokeUserRightsResponse*	Revoke rights from a user. Revoking rights does not affect the resource version of the corresponding user.
ListUserRights	*ListUserRightsRequest*	*ListUserRightsResponse*	List the set of all rights granted to a user.
UpdateUserIdentityProviderId	*UpdateUserIdentityProviderRequest*	*UpdateUserIdentityProviderResponse*	Update the assignment of a user from one IDP to another.

com/daml/ledger/api/v1/command_completion_service.proto

Checkpoint

Checkpoints may be used to:

detect time out of commands.
provide an offset which can be used to restart consumption.

Field	Type	Label	Description
record_time	google.protobuf.Timestamp		All commands with a maximum record time below this value MUST be considered lost if their completion has not arrived before this checkpoint. Required
offset	*LedgerOffset*		May be used in a subsequent CompletionStreamRequest to resume the consumption of this stream at a later time. Required

CompletionEndRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as described in `value.proto`). Optional

CompletionEndResponse

Field	Type	Label	Description
offset	*LedgerOffset*		This offset can be used in a CompletionStreamRequest message. Required

CompletionStreamRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger id reported by the Ledger Identification Service. Must be a valid LedgerString (as described in `value.proto`). Optional
application_id	*string*		Only completions of commands submitted with the same application_id will be visible in the stream. Must be a valid ApplicationIdString (as described in `value.proto`). Required unless authentication is used with a user token or a custom token specifying an application-id. In that case, the token's user-id, respectively application-id, will be used for the request's application_id.
parties	*string*	repeated	Non-empty list of parties whose data should be included. Only completions of commands for which at least one of the `act_as` parties is in the given set of parties will be visible in the stream. Must be a valid PartyIdString (as described in `value.proto`). Required
offset	*LedgerOffset*		This field indicates the minimum offset for completions. This can be used to resume an earlier completion stream. This offset is exclusive: the response will only contain commands whose offset is strictly greater than this. Optional, if not set the ledger uses the current ledger end offset instead.

CompletionStreamResponse

Field	Type	Label	Description
checkpoint	*Checkpoint*		This checkpoint may be used to restart consumption. The checkpoint is after any completions in this response. Optional
completions	*Completion*	repeated	If set, one or more completions.

CommandCompletionService

Allows clients to observe the status of their submissions. Commands may be submitted via the Command Submission Service. The on-ledger effects of their submissions are disclosed by the Transaction Service.

Commands may fail in 2 distinct manners:

1. Failure communicated synchronously in the gRPC error of the submission.
2. Failure communicated asynchronously in a Completion, see `completion.proto`.

Note that not only successfully submitted commands MAY produce a completion event. For example, the participant MAY choose to produce a completion event for a rejection of a duplicate command.

Clients that do not receive a successful completion about their submission MUST NOT assume that it was successful. Clients SHOULD subscribe to the CompletionStream before starting to submit commands to prevent race conditions.

Method name	Request type	Response type	Description
CompletionStream	*CompletionStreamRequest*	*CompletionStreamResponse*	Subscribe to command completion events.
CompletionEnd	*CompletionEndRequest*	*CompletionEndResponse*	Returns the offset after the latest completion.

com/daml/ledger/api/v1/command_service.proto

SubmitAndWaitForTransactionIdResponse

Field	Type	Label	Description
transaction_id	*string*		The id of the transaction that resulted from the submitted command. Must be a valid LedgerString (as described in `value.proto`). Required
completion_offset	*string*		The format of this field is described in `ledger_offset.proto`. Optional

SubmitAndWaitForTransactionResponse

Field	Type	Label	Description
transaction	*Transaction*		The flat transaction that resulted from the submitted command. Required
completion_offset	*string*		The format of this field is described in `ledger_offset.proto`. Optional

SubmitAndWaitForTransactionTreeResponse

Field	Type	Label	Description
transaction	*TransactionTree*		The transaction tree that resulted from the submitted command. Required
completion_offset	*string*		The format of this field is described in `ledger_offset.proto`. Optional

SubmitAndWaitRequest

These commands are atomic, and will become transactions.

Field	Type	Label	Description
commands	*Commands*		The commands to be submitted. Required

CommandService

Command Service is able to correlate submitted commands with completion data, identify timeouts, and return contextual information with each tracking result. This supports the implementation of stateless clients.

Note that submitted commands generally produce completion events as well, even in case a command gets rejected. For example, the participant MAY choose to produce a completion event for a rejection of a duplicate command.

Method name	Request type	Response type	Description
SubmitAndWait	*SubmitAndWaitRequest*	.google.protobuf.Empty	Submits a single composite command and waits for its result. Propagates the gRPC error of failed submissions including Daml interpretation errors.
SubmitAndWaitForTransactionId	*SubmitAndWaitRequest*	*SubmitAndWaitForTransactionIdResponse*	Submits a single composite command, waits for its result, and returns the transaction id. Propagates the gRPC error of failed submissions including Daml interpretation errors.
SubmitAndWaitForTransaction	*SubmitAndWaitRequest*	*SubmitAndWaitForTransactionResponse*	Submits a single composite command, waits for its result, and returns the transaction. Propagates the gRPC error of failed submissions including Daml interpretation errors.
SubmitAndWaitForTransactionTree	*SubmitAndWaitRequest*	*SubmitAndWaitForTransactionTreeResponse*	Submits a single composite command, waits for its result, and returns the transaction tree. Propagates the gRPC error of failed submissions including Daml interpretation errors.

com/daml/ledger/api/v1/command_submission_service.proto

SubmitRequest

The submitted commands will be processed atomically in a single transaction. Moreover, each `Command` in `commands` will be executed in the order specified by the request.

Field	Type	Label	Description
commands	*Commands*		The commands to be submitted in a single transaction. Required

CommandSubmissionService

Allows clients to attempt advancing the ledger's state by submitting commands. The final states of their submissions are disclosed by the Command Completion Service. The on-ledger effects of their submissions are disclosed by the Transaction Service.

Commands may fail in 2 distinct manners:

1. Failure communicated synchronously in the gRPC error of the submission.
2. Failure communicated asynchronously in a Completion, see `completion.proto`.

Note that not only successfully submitted commands MAY produce a completion event. For example, the participant MAY choose to produce a completion event for a rejection of a duplicate command.

Clients that do not receive a successful completion about their submission MUST NOT assume that it was successful. Clients SHOULD subscribe to the CompletionStream before starting to submit commands to prevent race conditions.

Method name	Request type	Response type	Description
Submit	*SubmitRequest*	.google.protobuf.Empty	Submit a single composite command.

com/daml/ledger/api/v1/commands.proto

Command

A command can either create a new contract or exercise a choice on an existing contract.

Field	Type	Label	Description
oneof command.create	*CreateCommand*		
oneof command.exercise	*ExerciseCommand*		
oneof command.exerciseByKey	*ExerciseByKeyCommand*		
oneof command.createAndExercise	*CreateAndExerciseCommand*		

Commands

A composite command that groups multiple commands together.

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as described in `value.proto`). Optional
workflow_id	*string*		Identifier of the on-ledger workflow that this command is a part of. Must be a valid LedgerString (as described in `value.proto`). Optional
application_id	*string*		Uniquely identifies the application or participant user that issued the command. Must be a valid ApplicationIdString (as described in `value.proto`). Required unless authentication is used with a user token or a custom token specifying an application-id. In that case, the token's user-id, respectively application-id, will be used for the request's application_id.
command_id	*string*		Uniquely identifies the command. The triple (application_id, party + act_as, command_id) constitutes the change ID for the intended ledger change, where party + act_as is interpreted as a set of party names. The change ID can be used for matching the intended ledger changes with all their completions. Must be a valid LedgerString (as described in `value.proto`). Required
party	*string*		Party on whose behalf the command should be executed. If ledger API authorization is enabled, then the authorization metadata must authorize the sender of the request to act on behalf of the given party. Must be a valid PartyIdString (as described in `value.proto`). Deprecated in favor of the `act_as` field. If both are set, then the effective list of parties on whose behalf the command should be executed is the union of all parties listed in `party` and `act_as`. Optional
commands	*Command*	repeated	Individual elements of this atomic command. Must be non-empty. Required
oneof deduplication_period.deduplication_time	google.protobuf.Duration		Specifies the length of the deduplication period. Same semantics apply as for *deduplication_duration*. Must be non-negative. Must not exceed the maximum deduplication time (see `ledger_configuration_service.proto`).
oneof deduplication_period.deduplication_duration	google.protobuf.Duration		Specifies the length of the deduplication period. It is interpreted relative to the local clock at some point during the submission's processing. Must be non-negative. Must not exceed the maximum deduplication time (see `ledger_configuration_service.proto`).
oneof deduplication_period.deduplication_offset	*string*		Specifies the start of the deduplication period by a completion stream offset (exclusive). Must be a valid LedgerString (as described in `ledger_offset.proto`).
min_ledger_time_abs	google.protobuf.Timestamp		Lower bound for the ledger time assigned to the resulting transaction. Note: The ledger time of a transaction is assigned as part of command interpretation. Use this prop-

If omitted, the participant or the committer may set a value of their choice. Optional

- disclosed_contracts
- *DisclosedContract*
- repeated
- Additional contracts used to resolve contract & contract key lookups. Optional

CreateAndExerciseCommand

Create a contract and exercise a choice on it in the same transaction.

Field	Type	Label	Description
template_id	*Identifier*		The template of the contract the client wants to create. Required
create_arguments	*Record*		The arguments required for creating a contract from this template. Required
choice	*string*		The name of the choice the client wants to exercise. Must be a valid NameString (as described in `value.proto`). Required
choice_argument	*Value*		The argument for this choice. Required

CreateCommand

Create a new contract instance based on a template.

Field	Type	Label	Description
template_id	*Identifier*		The template of contract the client wants to create. Required
create_arguments	*Record*		The arguments required for creating a contract from this template. Required

DisclosedContract

An additional contract that is used to resolve contract & contract key lookups.

Field	Type	Label	Description
template_id	*Identifier*		The template id of the contract. Required
contract_id	*string*		The contract id Required
oneof arguments.create_arguments	*Record*		The contract arguments as typed Record
oneof arguments.create_arguments_blob	google.protobuf.Any		The contract arguments specified using an opaque blob extracted from the `create_arguments_blob` field of a `com.daml.ledger.api.v1.CreatedEvent`.
metadata	*ContractMetadata*		The contract metadata from the create event. Required

ExerciseByKeyCommand

Exercise a choice on an existing contract specified by its key.

Field	Type	Label	Description
template_id	*Identifier*		The template of contract the client wants to exercise. Required
contract_key	*Value*		The key of the contract the client wants to exercise upon. Required
choice	*string*		The name of the choice the client wants to exercise. Must be a valid NameString (as described in `value.proto`) Required
choice_argument	*Value*		The argument for this choice. Required

ExerciseCommand

Exercise a choice on an existing contract.

Field	Type	Label	Description
template_id	*Identifier*		The template of contract the client wants to exercise. Required
contract_id	*string*		The ID of the contract the client wants to exercise upon. Must be a valid LedgerString (as described in `value.proto`). Required
choice	*string*		The name of the choice the client wants to exercise. Must be a valid NameString (as described in `value.proto`) Required
choice_argument	*Value*		The argument for this choice. Required

com/daml/ledger/api/v1/completion.proto

Completion

A completion represents the status of a submitted command on the ledger: it can be successful or failed.

Field	Type	Label	Description
command_id	*string*		The ID of the succeeded or failed command. Must be a valid LedgerString (as described in `value.proto`). Required
status	google.rpc.Status		Identifies the exact type of the error. It uses the same format of conveying error details as it is used for the RPC responses of the APIs. Optional
transaction_id	*string*		The transaction_id of the transaction that resulted from the command with command_id. Only set for successfully executed commands. Must be a valid LedgerString (as described in `value.proto`). Optional
application_id	*string*		The application-id or user-id that was used for the submission, as described in `commands.proto`. Must be a valid ApplicationIdString (as described in `value.proto`). Optional for historic completions where this data is not available.
act_as	*string*	repeated	The set of parties on whose behalf the commands were executed. Contains the union of `party` and `act_as` from `commands.proto`. The order of the parties need not be the same as in the submission. Each element must be a valid PartyIdString (as described in `value.proto`). Optional for historic completions where this data is not available.
submission_id	*string*		The submission ID this completion refers to, as described in `commands.proto`. Must be a valid LedgerString (as described in `value.proto`). Optional
oneof deduplication_period.deduplication_offset	*string*		Specifies the start of the deduplication period by a completion stream offset (exclusive).

Must be a valid LedgerString (as described in `value.proto`).

- oneof deduplication_period.deduplication_duration
- google.protobuf.Duration
-
- Specifies the length of the deduplication period. It is measured in record time of completions.

Must be non-negative.

com/daml/ledger/api/v1/contract_metadata.proto

ContractMetadata

Contract-related metadata used in DisclosedContract (that can be included in command submission) or forwarded as part of the CreateEvent in Active Contract Set or Transaction streams.

Field	Type	Label	Description
created_at	google.protobuf.Timestamp		Ledger effective time of the transaction that created the contract. Required
contract_key_hash	*bytes*		Hash of the contract key if defined. Optional
driver_metadata	*bytes*		Driver-specific metadata. This is opaque and cannot be decoded. Optional

com/daml/ledger/api/v1/event.proto

ArchivedEvent

Records that a contract has been archived, and choices may no longer be exercised on it.

Field	Type	Label	Description
event_id	*string*		The ID of this particular event. Must be a valid LedgerString (as described in `value.proto`). Required
contract_id	*string*		The ID of the archived contract. Must be a valid LedgerString (as described in `value.proto`). Required
template_id	*Identifier*		The template of the archived contract. Required
witness_parties	*string*	repeated	The parties that are notified of this event. For an `ArchivedEvent`, these are the intersection of the stakeholders of the contract in question and the parties specified in the `TransactionFilter`. The stakeholders are the union of the signatories and the observers of the contract. Each one of its elements must be a valid PartyIdString (as described in `value.proto`). Required

CreatedEvent

Records that a contract has been created, and choices may now be exercised on it.

Field	Type	Label	Description
event_id	*string*		The ID of this particular event. Must be a valid LedgerString (as described in `value.proto`). Required
contract_id	*string*		The ID of the created contract. Must be a valid LedgerString (as described in `value.proto`). Required
template_id	*Identifier*		The template of the created contract. Required
contract_key	*Value*		The key of the created contract. This will be set if and only if `create_arguments` is set and `template_id` defines a contract key. Optional
create_arguments	*Record*		The arguments that have been used to create the contract. Set either: - if there was a party, which is in the `witness_parties` of this event, and for which an `InclusiveFilters` exists with the `template_id` of this event among the `template_ids`, - or if there was a party, which is in the `witness_parties` of this event, and for which a wildcard filter exists (`Filters` without `InclusiveFilters`, or with an `InclusiveFilters` with empty `template_ids` and empty `interface_filters`). Optional
create_arguments_blob	google.protobuf.Any		Opaque representation of contract payload intended for forwarding to an API server as a contract disclosed as part of a command submission. Optional
interface_views	*InterfaceView*	repeated	Interface views specified in the transaction filter. Includes an `InterfaceView` for each interface for which there is a `InterfaceFilter` with - its party in the `witness_parties` of this event, - and which is implemented by the template of this event, - and which has `include_interface_view` set. Optional
witness_parties	*string*	repeated	The parties that are notified of this event. When a `CreatedEvent` is returned as part of a transaction tree, this will include all the parties specified in the `TransactionFilter` that are informees of the event. If served as part of a flat transaction those will be limited to all parties specified in the `TransactionFilter` that are stakeholders of the contract (i.e. either signatories or observers). In case of v2 API: If the `CreatedEvent` is returned as part of an AssignedEvent, ActiveContract or IncompleteUnassigned (so the event is related to an assignment or unassignment): this will include all parties of the `TransactionFilter` that are stakeholders of the contract. Required
signatories	*string*	repeated	The signatories for this contract as specified by the template. Required
observers	*string*	repeated	The observers for this contract as specified explicitly by the template or implicitly as choice controllers. This field never contains parties that are signatories. Required
agreement_text	google.protobuf.StringValue		The agreement text of the contract. We use StringValue to properly reflect optionality on the wire for backwards compatibility. This is necessary since the empty string is an acceptable (and in fact the default) agreement text, but also the default string in protobuf. This means a newer client works with an older sandbox seamlessly. Optional
	Contract		Metadata of the contract. Required for contracts created

Event

An event in the flat transaction stream can either be the creation or the archiving of a contract.

In the transaction service the events are restricted to the events visible for the parties specified in the transaction filter. Each event message type below contains a `witness_parties` field which indicates the subset of the requested parties that can see the event in question. In the flat transaction stream you'll only receive events that have witnesses.

Field	Type	Label	Description
oneof event.created	*CreatedEvent*		
oneof event.archived	*ArchivedEvent*		

ExercisedEvent

Records that a choice has been exercised on a target contract.

Field	Type	Label	Description
event_id	*string*		The ID of this particular event. Must be a valid LedgerString (as described in `value.proto`). Required
contract_id	*string*		The ID of the target contract. Must be a valid LedgerString (as described in `value.proto`). Required
template_id	*Identifier*		The template of the target contract. Required
interface_id	*Identifier*		The interface where the choice is defined, if inherited. Optional
choice	*string*		The choice that was exercised on the target contract. Must be a valid NameString (as described in `value.proto`). Required
choice_argument	*Value*		The argument of the exercised choice. Required
acting_parties	*string*	repeated	The parties that exercised the choice. Each element must be a valid PartyIdString (as described in `value.proto`). Required
consuming	*bool*		If true, the target contract may no longer be exercised. Required
witness_parties	*string*	repeated	The parties that are notified of this event. The witnesses of an exercise node will depend on whether the exercise was consuming or not. If consuming, the witnesses are the union of the stakeholders and the actors. If not consuming, the witnesses are the union of the signatories and the actors. Note that the actors might not necessarily be observers and thus signatories. This is the case when the controllers of a choice are specified using flexible controllers , using the `choice ... controller` syntax, and said controllers are not explicitly marked as observers. Each element must be a valid PartyIdString (as described in `value.proto`). Required
child_event_ids	*string*	repeated	References to further events in the same transaction that appeared as a result of this `ExercisedEvent`. It contains only the immediate children of this event, not all members of the subtree rooted at this node. The order of the children is the same as the event order in the transaction. Each element must be a valid LedgerString (as described in `value.proto`). Optional
exercise_result	*Value*		The result of exercising the choice. Required

InterfaceView

View of a create event matched by an interface filter.

Field	Type	Label	Description
interface_id	*Identifier*		The interface implemented by the matched event. Required
view_status	google.rpc.Status		Whether the view was successfully computed, and if not, the reason for the error. The error is reported using the same rules for error codes and messages as the errors returned for API requests. Required
view_value	*Record*		The value of the interface's view method on this event. Set if it was requested in the `InterfaceFilter` and it could be sucessfully computed. Optional

com/daml/ledger/api/v1/event_query_service.proto

GetEventsByContractIdRequest

Field	Type	Label	Description
contract_id	*string*		The contract id being queried. Required
requesting_parties	*string*	repeated	The parties whose events the client expects to see. The events associated with the contract id will only be returned if the requesting parties includes at least one party that is a stakeholder of the event. For a definition of stakeholders see https://docs.daml.com/concepts/ledger-model/ledger-privacy.html#contract-observers-and-stakeholders Required

GetEventsByContractIdResponse

Field	Type	Label	Description
create_event	*CreatedEvent*		The create event for the contract with the `contract_id` given in the request provided it exists and has not yet been pruned. Optional
archive_event	*ArchivedEvent*		The archive event for the contract with the `contract_id` given in the request provided such an archive event exists and it has not yet been pruned. Optional

GetEventsByContractKeyRequest

Field	Type	Label	Description
contract_key	*Value*		The contract key to search for. Required
template_id	*Identifier*		The template id associated with the contract key Required
requesting_parties	*string*	repeated	The parties whose events the client expects to see. The events associated with the contract key will only be returned if the requesting parties includes at least one party that is a stakeholder of the event. For a definition of stakeholders see https://docs.daml.com/concepts/ledger-model/ledger-privacy.html#contract-observers-and-stakeholders To gain visibility of all contract key bindings and to ensure consistent performance use a key maintainer as a requesting party. Required
continuation_token	*string*		A `continuation_token` associated with a previous response. Optional

GetEventsByContractKeyResponse

Field	Type	Label	Description
create_event	*CreatedEvent*		The most recent create event for a contract with the key given in the request, if no `continuation_token` is provided. If a `continuation_token` is provided, then this is the most recent create event preceding the create event whose `continuation_token` was provided. Optional
archive_event	*ArchivedEvent*		The archive event for the create event provided the created contract is archived. Optional
continuation_token	*string*		If the `continuation_token` is populated then there may be additional events available. To retrieve these events use the `continuation_token` in a subsequent request. Optional

EventQueryService

Query events by contract id or key.

Method name	Request type	Response type	Description
GetEventsByContractId	*GetEventsByContractIdRequest*	*GetEventsByContractIdResponse*	Get the create and the consuming exercise event for the contract with the provided ID. No events will be returned for contracts that have been pruned because they have already been archived before the latest pruning offset.
GetEventsByContractKey	*GetEventsByContractKeyRequest*	*GetEventsByContractKeyResponse*	Get all create and consuming exercise events for the contracts with the provided contract key. Only events for unpruned contracts will be returned. Matching events are delivered in reverse chronological order, i.e., the most recent events are delivered first.

com/daml/ledger/api/v1/experimental_features.proto

AcsActiveAtOffsetFeature

Whether the Ledger API supports requesting ACS at an offset

Field	Type	Label	Description
supported	*bool*		

CommandDeduplicationFeatures

Feature descriptors for command deduplication intended to be used for adapting Ledger API tests.

Field	Type	Label	Description
deduplication_period_support	*CommandDeduplicationPeriodSupport*		
deduplication_type	*CommandDeduplicationType*		
max_deduplication_duration_enforced	*bool*		The ledger will reject any requests which specify a deduplication period which exceeds the specified max deduplication duration. This is also enforced for ledgers that convert deduplication periods specified as offsets to durations.

CommandDeduplicationPeriodSupport

Feature descriptor specifying how deduplication periods can be specified and how they are handled by the participant node.

Field	Type	Label	Description
offset_support	*CommandDeduplicationPeriodSupport.OffsetSupport*		
duration_support	*CommandDeduplicationPeriodSupport.DurationSupport*		

ExperimentalCommitterEventLog

How the committer stores events.

Field	Type	Label	Description
event_log_type	*ExperimentalCommitterEventLog.CommitterEventLogType*		

ExperimentalContractIds

See *daml-lf/spec/contract-id.rst* for more information on contract ID formats.

Field	Type	Label	Description
v1	*ExperimentalContractIds.ContractIdV1Support*		

ExperimentalExplicitDisclosure

Enables the use of explicitly disclosed contracts for command submission

Field	Type	Label	Description
supported	*bool*		

ExperimentalFeatures

See the feature message definitions for descriptions.

Field	Type	Label	Description
self_service_error_codes	*ExperimentalSelfServiceErrorCodes*		
static_time	*ExperimentalStaticTime*		
command_deduplication	*CommandDeduplicationFeatures*		
optional_ledger_id	*ExperimentalOptionalLedgerId*		
contract_ids	*ExperimentalContractIds*		
committer_event_log	*ExperimentalCommitterEventLog*		
explicit_disclosure	*ExperimentalExplicitDisclosure*		
user_and_party_local_metadata_extensions	*ExperimentalUserAndPartyLocalMetadataExtensions*		
acs_active_at_offset	*AcsActiveAtOffsetFeature*		

ExperimentalOptionalLedgerId

Ledger API does not require ledgerId to be set in the requests.

ExperimentalSelfServiceErrorCodes

GRPC self-service error codes are returned by the Ledger API.

ExperimentalStaticTime

Ledger is in the static time mode and exposes a time service.

Field	Type	Label	Description
supported	*bool*		

ExperimentalUserAndPartyLocalMetadataExtensions

Whether the Ledger API supports: - is_deactivated user property, - metadata with annotations and resource version for users and parties, - update calls for users and parties.

Field	Type	Label	Description
supported	*bool*		

CommandDeduplicationPeriodSupport.DurationSupport

How the participant node supports deduplication periods specified as durations.

Name	Number	Description
DURATION_NATIVE_SUPPORT	0	
DURATION_CONVERT_TO_OFFSET	1	

CommandDeduplicationPeriodSupport.OffsetSupport

How the participant node supports deduplication periods specified using offsets.

Name	Number	Description
OFFSET_NOT_SUPPORTED	0	
OFFSET_NATIVE_SUPPORT	1	
OFFSET_CONVERT_TO_DURATION	2	

CommandDeduplicationType

How the participant node reports duplicate command submissions.

Name	Number	Description
ASYNC_ONLY	0	Duplicate commands are exclusively reported asynchronously via completions.
ASYNC_AND_CONCURRENT_SYNC	1	Commands that are duplicates of concurrently submitted commands are reported synchronously via a gRPC error on the command submission, while all other duplicate commands are reported asynchronously via completions.

ExperimentalCommitterEventLog.CommitterEventLogType

Name	Number	Description
CENTRALIZED	0	Default. There is a single log.
DISTRIBUTED	1	There is more than one event log. Usually, when the committer itself is distributed. Or there are per-participant event logs. It may result in transaction IDs being different for the same transaction across participants, for example.

ExperimentalContractIds.ContractIdV1Support

Name	Number	Description
SUFFIXED	0	Contract IDs must be suffixed. Distributed ledger implementations must reject non-suffixed contract IDs.
NON_SUFFIXED	1	Contract IDs do not need to be suffixed. This can be useful for shorter contract IDs in centralized committer implementations. Suffixed contract IDs must also be supported.

com/daml/ledger/api/v1/ledger_configuration_service.proto

GetLedgerConfigurationRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as described in `value.proto`). Optional

GetLedgerConfigurationResponse

Field	Type	Label	Description
ledger_configuration	*LedgerConfiguration*		The latest ledger configuration.

LedgerConfiguration

LedgerConfiguration contains parameters of the ledger instance that may be useful to clients.

Field	Type	Label	Description
max_deduplication_duration	google.protobuf.Duration		If a command submission specifies a deduplication period of length up to `max_deduplication_duration`, the submission SHOULD not be rejected with `FAILED_PRECONDITION` because the deduplication period starts too early. The deduplication period is measured on a local clock of the participant or Daml ledger, and therefore subject to clock skews and clock drifts. Command submissions with longer periods MAY get accepted though.

LedgerConfigurationService

LedgerConfigurationService allows clients to subscribe to changes of the ledger configuration. In V2 Ledger API this service is not available anymore.

Method name	Request type	Response type	Description
GetLedgerConfiguration	*GetLedgerConfigurationRequest*	*GetLedgerConfigurationResponse*	Returns the latest configuration as the first response, and publishes configuration updates in the same stream.

com/daml/ledger/api/v1/ledger_identity_service.proto

GetLedgerIdentityRequest

GetLedgerIdentityResponse

Field	Type	Label	Description
ledger_id	*string*		The ID of the ledger exposed by the server. Must be a valid LedgerString (as described in `value.proto`). Optional

LedgerIdentityService

DEPRECATED: This service is now deprecated and ledger identity string is optional for all Ledger API requests.

Allows clients to verify that the server they are communicating with exposes the ledger they wish to operate on. In V2 Ledger API this service is not available anymore.

Method name	Request type	Response type	Description
GetLedgerIdentity	*GetLedgerIdentityRequest*	*GetLedgerIdentityResponse*	Clients may call this RPC to return the identifier of the ledger they are connected to.

com/daml/ledger/api/v1/ledger_offset.proto

LedgerOffset

Describes a specific point on the ledger.

The Ledger API endpoints that take offsets allow to specify portions of the ledger that are relevant for the client to read.

Offsets returned by the Ledger API can be used as-is (e.g. to keep track of processed transactions and provide a restart point to use in case of need).

The format of absolute offsets is opaque to the client: no client-side transformation of an offset is guaranteed to return a meaningful offset.

The server implementation ensures internally that offsets are lexicographically comparable.

Field	Type	Label	Description
oneof value.absolute	*string*		The format of this string is specific to the ledger and opaque to the client.
oneof value.boundary	*LedgerOffset.LedgerBoundary*		

LedgerOffset.LedgerBoundary

Name	Number	Description
LEDGER_BEGIN	0	Refers to the first transaction.
LEDGER_END	1	Refers to the currently last transaction, which is a moving target.

com/daml/ledger/api/v1/package_service.proto

GetPackageRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as described in `value.proto`). Optional
package_id	*string*		The ID of the requested package. Must be a valid PackageIdString (as described in `value.proto`). Required

GetPackageResponse

Field	Type	Label	Description
hash_function	*HashFunction*		The hash function we use to calculate the hash. Required
archive_payload	*bytes*		Contains a `daml_lf` ArchivePayload. See further details in `daml_lf.proto`. Required
hash	*string*		The hash of the archive payload, can also used as a `package_id`. Must be a valid PackageIdString (as described in `value.proto`). Required

GetPackageStatusRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as described in `value.proto`). Optional
package_id	*string*		The ID of the requested package. Must be a valid PackageIdString (as described in `value.proto`). Required

GetPackageStatusResponse

Field	Type	Label	Description
package_status	*PackageStatus*		The status of the package.

ListPackagesRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as described in `value.proto`). Optional

ListPackagesResponse

Field	Type	Label	Description
package_ids	*string*	repeated	The IDs of all Daml-LF packages supported by the server. Each element must be a valid PackageIdString (as described in `value.proto`). Required

HashFunction

Name	Number	Description
SHA256	0	

PackageStatus

Name	Number	Description
UNKNOWN	0	The server is not aware of such a package.
REGISTERED	1	The server is able to execute Daml commands operating on this package.

PackageService

Allows clients to query the Daml-LF packages that are supported by the server.

Method name	Request type	Response type	Description
ListPackages	*ListPackagesRequest*	*ListPackagesResponse*	Returns the identifiers of all supported packages.
GetPackage	*GetPackageRequest*	*GetPackageResponse*	Returns the contents of a single package.
GetPackageStatus	*GetPackageStatusRequest*	*GetPackageStatusResponse*	Returns the status of a single package.

com/daml/ledger/api/v1/testing/time_service.proto

GetTimeRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as describe in `value.proto`). Optional

GetTimeResponse

Field	Type	Label	Description
current_time	google.protobuf.Timestamp		The current time according to the ledger server.

SetTimeRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as describe in `value.proto`). Optional
current_time	google.protobuf.Timestamp		MUST precisely match the current time as it's known to the ledger server.
new_time	google.protobuf.Timestamp		The time the client wants to set on the ledger. MUST be a point int time after `current_time`.

TimeService

Optional service, exposed for testing static time scenarios.

Method name	Request type	Response type	Description
GetTime	*GetTimeRequest*	*GetTimeResponse*	Returns a stream of time updates. Always returns at least one response, where the first one is the current time. Subsequent responses are emitted whenever the ledger server's time is updated.
SetTime	*SetTimeRequest*	.google.protobuf.Empty	Allows clients to change the ledger's clock in an atomic get-and-set operation.

com/daml/ledger/api/v1/transaction.proto

Transaction

Filtered view of an on-ledger transaction's create and archive events.

Field	Type	Label	Description
transaction_id	*string*		Assigned by the server. Useful for correlating logs. Must be a valid LedgerString (as described in `value.proto`). Required
command_id	*string*		The ID of the command which resulted in this transaction. Missing for everyone except the submitting party. Must be a valid LedgerString (as described in `value.proto`). Optional
workflow_id	*string*		The workflow ID used in command submission. Must be a valid LedgerString (as described in `value.proto`). Optional
effective_at	google.protobuf.Timestamp		Ledger effective time. Must be a valid LedgerString (as described in `value.proto`). Required
events	*Event*	repeated	The collection of events. Only contains `CreatedEvent` or `ArchivedEvent`. Required
offset	*string*		The absolute offset. The format of this field is described in `ledger_offset.proto`. Required

TransactionTree

Complete view of an on-ledger transaction.

Field	Type	Label	Description
transaction_id	*string*		Assigned by the server. Useful for correlating logs. Must be a valid LedgerString (as described in `value.proto`). Required
command_id	*string*		The ID of the command which resulted in this transaction. Missing for everyone except the submitting party. Must be a valid LedgerString (as described in `value.proto`). Optional
workflow_id	*string*		The workflow ID used in command submission. Only set if the `workflow_id` for the command was set. Must be a valid LedgerString (as described in `value.proto`). Optional
effective_at	google.protobuf.Timestamp		Ledger effective time. Required
offset	*string*		The absolute offset. The format of this field is described in `ledger_offset.proto`. Required
events_by_id	*TransactionTree.EventsByIdEntry*	repeated	Changes to the ledger that were caused by this transaction. Nodes of the transaction tree. Each key be a valid LedgerString (as describe in `value.proto`). Required
root_event_ids	*string*	repeated	Roots of the transaction tree. Each element must be a valid LedgerString (as describe in `value.proto`). The elements are in the same order as the commands in the corresponding Commands object that triggered this transaction. Required

TransactionTree.EventsByIdEntry

Field	Type	Label	Description
key	*string*		
value	*TreeEvent*		

TreeEvent

Each tree event message type below contains a `witness_parties` field which indicates the subset of the requested parties that can see the event in question.

Note that transaction trees might contain events with _no_ witness parties, which were included simply because they were children of events which have witnesses.

Field	Type	Label	Description
oneof kind.created	*CreatedEvent*		
oneof kind.exercised	*ExercisedEvent*		

com/daml/ledger/api/v1/transaction_filter.proto

Filters

The union of a set of contract filters, or a wildcard.

Field	Type	Label	Description
inclusive	*InclusiveFilters*		If set, then contracts matching any of the `InclusiveFilters` match this filter. If not set, or if `InclusiveFilters` has empty `template_ids` and empty `interface_filters`: any contract matches this filter. Optional

InclusiveFilters

A filter that matches all contracts that are either an instance of one of the `template_ids` or that match one of the `interface_filters`.

Field	Type	Label	Description
template_ids	*Identifier*	repeated	A collection of templates for which the payload will be included in the `create_arguments` of a matching `CreatedEvent`. SHOULD NOT contain duplicates. All `template_ids` needs to be valid: corresponding template should be defined in one of the available packages at the time of the query. Optional
interface_filters	*InterfaceFilter*	repeated	Include an `InterfaceView` for every `InterfaceFilter` matching a contract. The `InterfaceFilter``s MUST use unique ``interface_id``s. All ``interface_id` needs to be valid: corresponding interface should be defined in one of the available packages at the time of the query. Optional

InterfaceFilter

This filter matches contracts that implement a specific interface.

Field	Type	Label	Description
interface_id	*Identifier*		The interface that a matching contract must implement. Required
include_interface_view	*bool*		Whether to include the interface view on the contract in the returned `CreateEvent`. Use this to access contract data in a uniform manner in your API client. Optional
include_create_arguments_blob	*bool*		Whether to include a `create_arguments_blob` in the returned `CreateEvent`. Use this to access the complete contract data in your API client for submitting it as a disclosed contract with future commands. Optional

TransactionFilter

A filter both for filtering create and archive events as well as for filtering transaction trees.

Field	Type	Label	Description
filters_by_party	*TransactionFilter.FiltersByPartyEntry*	repeated	Each key must be a valid PartyIdString (as described in `value.proto`). The interpretation of the filter depends on the stream being filtered: (1) For **transaction tree streams** only party filters with wildcards are allowed, and all subtrees whose root has one of the listed parties as an informee are returned. (2) For **transaction and active-contract-set streams** create and archive events are returned for all contracts whose stakeholders include at least one of the listed parties and match the per-party filter. Required

TransactionFilter.FiltersByPartyEntry

Field	Type	Label	Description
key	*string*		
value	*Filters*		

com/daml/ledger/api/v1/transaction_service.proto

GetFlatTransactionResponse

Field	Type	Label	Description
transaction	*Transaction*		

GetLatestPrunedOffsetsRequest

Empty for now, but may contain fields in the future.

GetLatestPrunedOffsetsResponse

Field	Type	Label	Description
participant_pruned_up_to_inclusive	*LedgerOffset*		The offset up to which the ledger has been pruned, disregarding the state of all divulged contracts pruning.
all_divulged_contracts_pruned_up_to_inclusive	*LedgerOffset*		The offset up to which all divulged events have been pruned on the ledger. It can be at or before the `participant_pruned_up_to_inclusive` offset. For more details about all divulged events pruning, see `PruneRequest.prune_all_divulged_contracts` in `participant_pruning_service.proto`.

GetLedgerEndRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as describe in `value.proto`). Optional

GetLedgerEndResponse

Field	Type	Label	Description
offset	*LedgerOffset*		The absolute offset of the current ledger end.

GetTransactionByEventIdRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as described in `value.proto`). Optional
event_id	*string*		The ID of a particular event. Must be a valid LedgerString (as described in `value.proto`). Required
requesting_parties	*string*	repeated	The parties whose events the client expects to see. Events that are not visible for the parties in this collection will not be present in the response. Each element must be a valid PartyIdString (as described in `value.proto`). Required

GetTransactionByIdRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as describe in `value.proto`). Optional
transaction_id	*string*		The ID of a particular transaction. Must be a valid LedgerString (as describe in `value.proto`). Required
requesting_parties	*string*	repeated	The parties whose events the client expects to see. Events that are not visible for the parties in this collection will not be present in the response. Each element be a valid PartyIdString (as describe in `value.proto`). Required

GetTransactionResponse

Field	Type	Label	Description
transaction	*TransactionTree*		

GetTransactionTreesResponse

Field	Type	Label	Description
transactions	*TransactionTree*	repeated	The list of transaction trees that matches the filter in `GetTransactionsRequest` for the `GetTransactionTrees` method.

GetTransactionsRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as described in `value.proto`). Optional
begin	*LedgerOffset*		Beginning of the requested ledger section. This offset is exclusive: the response will only contain transactions whose offset is strictly greater than this. Required
end	*LedgerOffset*		End of the requested ledger section. This offset is inclusive: the response will only contain transactions whose offset is less than or equal to this. Optional, if not set, the stream will not terminate.
filter	*TransactionFilter*		Requesting parties with template filters. Template filters must be empty for GetTransactionTrees requests. Required
verbose	*bool*		If enabled, values served over the API will contain more information than strictly necessary to interpret the data. In particular, setting the verbose flag to true triggers the ledger to include labels for record fields. Optional

GetTransactionsResponse

Field	Type	Label	Description
transactions	*Transaction*	repeated	The list of transactions that matches the filter in GetTransactionsRequest for the GetTransactions method.

TransactionService

Allows clients to read transactions from the ledger. In V2 Ledger API this service is not available anymore. Use v2.UpdateService instead.

Method name	Request type	Response type	Description
GetTransactions	*GetTransactionsRequest*	*GetTransactionsResponse*	Read the ledger's filtered transaction stream for a set of parties. Lists only creates and archives, but not other events. Omits all events on transient contracts, i.e., contracts that were both created and archived in the same transaction.
GetTransactionTrees	*GetTransactionsRequest*	*GetTransactionTreesResponse*	Read the ledger's complete transaction tree stream for a set of parties. The stream can be filtered only by parties, but not templates (template filter must be empty).
GetTransactionByEventId	*GetTransactionByEventIdRequest*	*GetTransactionResponse*	Lookup a transaction tree by the ID of an event that appears within it. For looking up a transaction instead of a transaction tree, please see GetFlatTransactionByEventId
GetTransactionById	*GetTransactionByIdRequest*	*GetTransactionResponse*	Lookup a transaction tree by its ID. For looking up a transaction instead of a transaction tree, please see GetFlatTransactionById
GetFlatTransactionByEventId	*GetTransactionByEventIdRequest*	*GetFlatTransactionResponse*	Lookup a transaction by the ID of an event that appears within it.
GetFlatTransactionById	*GetTransactionByIdRequest*	*GetFlatTransactionResponse*	Lookup a transaction by its ID.
GetLedgerEnd	*GetLedgerEndRequest*	*GetLedgerEndResponse*	Get the current ledger end. Subscriptions started with the returned offset will serve transactions created after this RPC was called.
GetLatestPrunedOffsets	*GetLatestPrunedOffsetsRequest*	*GetLatestPrunedOffsetsResponse*	Get the latest successfully pruned ledger offsets

com/daml/ledger/api/v1/value.proto

Enum

A value with finite set of alternative representations.

Field	Type	Label	Description
enum_id	*Identifier*		Omitted from the transaction stream when verbose streaming is not enabled. Optional when submitting commands.
constructor	*string*		Determines which of the Variant's alternatives is encoded in this message. Must be a valid NameString. Required

GenMap

Field	Type	Label	Description
entries	*GenMap.Entry*	repeated	

GenMap.Entry

Field	Type	Label	Description
key	*Value*		
value	*Value*		

Identifier

Unique identifier of an entity.

Field	Type	Label	Description
package_id	*string*		The identifier of the Daml package that contains the entity. Must be a valid PackageIdString. Required
module_name	*string*		The dot-separated module name of the identifier. Required
entity_name	*string*		The dot-separated name of the entity (e.g. record, template,) within the module. Required

List

A homogenous collection of values.

Field	Type	Label	Description
elements	*Value*	repeated	The elements must all be of the same concrete value type. Optional

Map

Field	Type	Label	Description
entries	*Map.Entry*	repeated	

Map.Entry

Field	Type	Label	Description
key	*string*		
value	*Value*		

Optional

Corresponds to Java's Optional type, Scala's Option, and Haskell's Maybe. The reason why we need to wrap this in an additional `message` is that we need to be able to encode the `None` case in the `Value` oneof.

Field	Type	Label	Description
value	*Value*		optional

Record

Contains nested values.

Field	Type	Label	Description
record_id	*Identifier*		Omitted from the transaction stream when verbose streaming is not enabled. Optional when submitting commands.
fields	*RecordField*	repeated	The nested values of the record. Required

RecordField

A named nested value within a record.

Field	Type	Label	Description
label	*string*		When reading a transaction stream, it's omitted if verbose streaming is not enabled. When submitting a commmand, it's optional: - if all keys within a single record are present, the order in which fields appear does not matter. however, each key must appear exactly once. - if any of the keys within a single record are omitted, the order of fields MUST match the order of declaration in the Daml template. Must be a valid NameString
value	*Value*		A nested value of a record. Required

Value

Encodes values that the ledger accepts as command arguments and emits as contract arguments.

The values encoding use different classes of non-empty strings as identifiers. Those classes are defined as follows: - NameStrings are strings with length <= 1000 that match the regexp `[A-Za-z\$_][A-Za-z0-9\$_]*`. - PackageIdStrings are strings with length <= 64 that match the regexp `[A-Za-z0-9\-_ ]+`. - PartyIdStrings are strings with length <= 255 that match the regexp `[A-Za-z0-9:\-_ ]+`. - LedgerStrings are strings with length <= 255 that match the regexp `[A-Za-z0-9#:\-_/ ]+`. - ApplicationIdStrings are strings with length <= 255 that match the regexp `[A-Za-z0-9#:\-_/ @\|]+`.

Field	Type	Label	Description
oneof Sum.record	*Record*		
oneof Sum.variant	*Variant*		
oneof Sum.contract_id	*string*		Identifier of an on-ledger contract. Commands which reference an unknown or already archived contract ID will fail. Must be a valid LedgerString.
oneof Sum.list	*List*		Represents a homogeneous list of values.
oneof Sum.int64	*sint64*		
oneof Sum.numeric	*string*		A Numeric, that is a decimal value with precision 38 (at most 38 significant digits) and a scale between 0 and 37 (significant digits on the right of the decimal point). The field has to match the regex [+-]?d{1,38}(.d{0,37})? and should be representable by a Numeric without loss of precision.
oneof Sum.text	*string*		A string.
oneof Sum.timestamp	*sfixed64*		Microseconds since the UNIX epoch. Can go backwards. Fixed since the vast majority of values will be greater than 2^28, since currently the number of microseconds since the epoch is greater than that. Range: 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999Z, so that we can convert to/from https://www.ietf.org/rfc/rfc3339.txt
oneof Sum.party	*string*		An agent operating on the ledger. Must be a valid PartyIdString.
oneof Sum.bool	*bool*		True or false.
oneof Sum.unit	google.protobuf.Empty		This value is used for example for choices that don't take any arguments.
oneof Sum.date	*int32*		Days since the unix epoch. Can go backwards. Limited from 0001-01-01 to 9999-12-31, also to be compatible with https://www.ietf.org/rfc/rfc3339.txt
oneof Sum.optional	*Optional*		The Optional type, None or Some
oneof Sum.map	*Map*		The Map type
oneof Sum.enum	*Enum*		The Enum type
oneof Sum.gen_map	*GenMap*		The GenMap type

Variant

A value with alternative representations.

Field	Type	Label	Description
variant_id	*Identifier*		Omitted from the transaction stream when verbose streaming is not enabled. Optional when submitting commands.
constructor	*string*		Determines which of the Variant's alternatives is encoded in this message. Must be a valid NameString. Required
value	*Value*		The value encoded within the Variant. Required

com/daml/ledger/api/v1/version_service.proto

FeaturesDescriptor

Field	Type	Label	Description
user_management	*UserManagementFeature*		If set, then the Ledger API server supports user management. It is recommended that clients query this field to gracefully adjust their behavior for ledgers that do not support user management.
experimental	*ExperimentalFeatures*		Features under development or features that are used for ledger implementation testing purposes only.

Daml applications SHOULD not depend on these in production.

GetLedgerApiVersionRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as described in `value.proto`). Optional

GetLedgerApiVersionResponse

Field	Type	Label	Description
version	*string*		The version of the ledger API.
features	*FeaturesDescriptor*		The features supported by this Ledger API endpoint.

Daml applications CAN use the feature descriptor on top of version constraints on the Ledger API version to determine whether a given Ledger API endpoint supports the features required to run the application.

See the feature descriptions themselves for the relation between Ledger API versions and feature presence.

UserManagementFeature

Field	Type	Label	Description
supported	*bool*		Whether the Ledger API server provides the user management service.
max_rights_per_user	*int32*		The maximum number of rights that can be assigned to a single user. Servers MUST support at least 100 rights per user. A value of 0 means that the server enforces no rights per user limit.
max_users_page_size	*int32*		The maximum number of users the server can return in a single response (page). Servers MUST support at least a 100 users per page. A value of 0 means that the server enforces no page size limit.

VersionService

Allows clients to retrieve information about the ledger API version

Method name	Request type	Response type	Description
GetLedgerApiVersion	*GetLedgerApiVersionRequest*	*GetLedgerApiVersionResponse*	Read the Ledger API version

com/daml/ledger/api/v2/command_completion_service.proto

CompletionStreamRequest

Field	Type	Label	Description
application_id	*string*		Only completions of commands submitted with the same application_id will be visible in the stream. Must be a valid ApplicationIdString (as described in `value.proto`). Required unless authentication is used with a user token or a custom token specifying an application-id. In that case, the token's user-id, respectively application-id, will be used for the request's application_id.
parties	*string*	repeated	Non-empty list of parties whose data should be included. Only completions of commands for which at least one of the `act_as` parties is in the given set of parties will be visible in the stream. Must be a valid PartyIdString (as described in `value.proto`). Required
begin_exclusive	*ParticipantOffset*		This field indicates the minimum offset for completions. This can be used to resume an earlier completion stream. Optional, if not set the ledger uses the current ledger end offset instead.

CompletionStreamResponse

Field	Type	Label	Description
checkpoint	*com.daml.ledger.api.v1.Checkpoint*		This checkpoint may be used to restart consumption. The checkpoint belongs to the completion in this response. Required
completion	*Completion*		Required
domain_id	*string*		The sequencing domain. In case - successful/failed transactions: identifies the sequencing domain of the transaction - for successful/failed unassign commands: identifies the source domain - for successful/failed assign commands: identifies the target domain Required

CommandCompletionService

Allows clients to observe the status of their submissions. Commands may be submitted via the Command Submission Service. The on-ledger effects of their submissions are disclosed by the Update Service.

Commands may fail in 2 distinct manners:

1. Failure communicated synchronously in the gRPC error of the submission.
2. Failure communicated asynchronously in a Completion, see `completion.proto`.

Note that not only successfully submitted commands MAY produce a completion event. For example, the participant MAY choose to produce a completion event for a rejection of a duplicate command.

Clients that do not receive a successful completion about their submission MUST NOT assume that it was successful. Clients SHOULD subscribe to the CompletionStream before starting to submit commands to prevent race conditions.

Method name	Request type	Response type	Description
CompletionStream	*CompletionStreamRequest*	*CompletionStreamResponse*	Subscribe to command completion events.

com/daml/ledger/api/v2/command_service.proto

SubmitAndWaitForTransactionResponse

Field	Type	Label	Description
transaction	*Transaction*		The flat transaction that resulted from the submitted command. Required
completion_offset	*string*		The format of this field is described in `participant_offset.proto`. Optional

SubmitAndWaitForTransactionTreeResponse

Field	Type	Label	Description
transaction	*TransactionTree*		The transaction tree that resulted from the submitted command. Required
completion_offset	*string*		The format of this field is described in `participant_offset.proto`. Optional

SubmitAndWaitForUpdateIdResponse

Field	Type	Label	Description
update_id	*string*		The id of the transaction that resulted from the submitted command. Must be a valid LedgerString (as described in `value.proto`). Required
completion_offset	*string*		The format of this field is described in `participant_offset.proto`. Optional

SubmitAndWaitRequest

These commands are atomic, and will become transactions.

Field	Type	Label	Description
commands	*Commands*		The commands to be submitted. Required

CommandService

Command Service is able to correlate submitted commands with completion data, identify timeouts, and return contextual information with each tracking result. This supports the implementation of stateless clients.

Note that submitted commands generally produce completion events as well, even in case a command gets rejected. For example, the participant SHOULD produce a completion event for a rejection of a duplicate command.

Method name	Request type	Response type	Description
SubmitAndWait	*SubmitAndWaitRequest*	.google.protobuf.Empty	Submits a single composite command and waits for its result. Propagates the gRPC error of failed submissions including Daml interpretation errors.
SubmitAndWaitForUpdateId	*SubmitAndWaitRequest*	*SubmitAndWaitForUpdateIdResponse*	Submits a single composite command, waits for its result, and returns the update id. Propagates the gRPC error of failed submissions including Daml interpretation errors.
SubmitAndWaitForTransaction	*SubmitAndWaitRequest*	*SubmitAndWaitForTransactionResponse*	Submits a single composite command, waits for its result, and returns the transaction. Propagates the gRPC error of failed submissions including Daml interpretation errors.
SubmitAndWaitForTransactionTree	*SubmitAndWaitRequest*	*SubmitAndWaitForTransactionTreeResponse*	Submits a single composite command, waits for its result, and returns the transaction tree. Propagates the gRPC error of failed submissions including Daml interpretation errors.

com/daml/ledger/api/v2/command_submission_service.proto

SubmitReassignmentRequest

Field	Type	Label	Description
reassignment_command	*ReassignmentCommand*		The reassignment command to be submitted. Required

SubmitReassignmentResponse

SubmitRequest

The submitted commands will be processed atomically in a single transaction. Moreover, each `Command` in `commands` will be executed in the order specified by the request.

Field	Type	Label	Description
commands	*Commands*		The commands to be submitted in a single transaction. Required

SubmitResponse

CommandSubmissionService

Allows clients to attempt advancing the ledger's state by submitting commands. The final states of their submissions are disclosed by the Command Completion Service. The on-ledger effects of their submissions are disclosed by the Update Service.

Commands may fail in 2 distinct manners:

1. Failure communicated synchronously in the gRPC error of the submission.
2. Failure communicated asynchronously in a Completion, see `completion.proto`.

Note that not only successfully submitted commands MAY produce a completion event. For example, the participant MAY choose to produce a completion event for a rejection of a duplicate command.

Clients that do not receive a successful completion about their submission MUST NOT assume that it was successful. Clients SHOULD subscribe to the CompletionStream before starting to submit commands to prevent race conditions.

Method name	Request type	Response type	Description
Submit	*SubmitRequest*	*SubmitResponse*	Submit a single composite command.
SubmitReassignment	*SubmitReassignmentRequest*	*SubmitReassignmentResponse*	Submit a single reassignment.

com/daml/ledger/api/v2/commands.proto

Commands

A composite command that groups multiple commands together.

Field	Type	Label	Description
workflow_id	*string*		Identifier of the on-ledger workflow that this command is a part of. Must be a valid LedgerString (as described in `value.proto`). Optional
application_id	*string*		Uniquely identifies the application or participant user that issued the command. Must be a valid ApplicationIdString (as described in `value.proto`). Required unless authentication is used with a user token or a custom token specifying an application-id. In that case, the token's user-id, respectively application-id, will be used for the request's application_id.
command_id	*string*		Uniquely identifies the command. The triple (application_id, party + act_as, command_id) constitutes the change ID for the intended ledger change, where party + act_as is interpreted as a set of party names. The change ID can be used for matching the intended ledger changes with all their completions. Must be a valid LedgerString (as described in `value.proto`). Required
party	*string*		Party on whose behalf the command should be executed. If ledger API authorization is enabled, then the authorization metadata must authorize the sender of the request to act on behalf of the given party. Must be a valid PartyIdString (as described in `value.proto`). Deprecated in favor of the `act_as` field. If both are set, then the effective list of parties on whose behalf the command should be executed is the union of all parties listed in `party` and `act_as`. Optional
commands	*com.daml.ledger.api.v1.Command*	repeated	Individual elements of this atomic command. Must be non-empty. Required
oneof deduplication_period.deduplication_duration	google.protobuf.Duration		Specifies the length of the deduplication period. It is interpreted relative to the local clock at some point during the submission's processing. Must be non-negative. Must not exceed the maximum deduplication time (see `ledger_configuration_service.proto`).
oneof deduplication_period.deduplication_offset	*string*		Specifies the start of the deduplication period by a completion stream offset (exclusive). Must be a valid LedgerString (as described in `participant_offset.proto`).
min_ledger_time_abs	google.protobuf.Timestamp		Lower bound for the ledger time assigned to the resulting transaction. Note: The ledger time of a transaction is assigned as part of command interpretation. Use this property if you expect that command interpretation will take a considerate amount of time, such that by the time the resulting transaction is sequenced, its assigned ledger time is not valid anymore. Must not be set at the same time as min_ledger_time_rel. Optional
min_ledger_time_rel	google.protobuf.Duration		Same as min_ledger_time_abs, but specified as a duration, starting from the time the command is received by the server. Must not be set at the same time as min_ledger_time_abs. Optional
string	*string*	repeated	Set of parties on whose behalf the command should be

If omitted, the participant or the committer may set a value of their choice. Optional

- disclosed_contracts
- *com.daml.ledger.api.v1.DisclosedContract*
- repeated
- Additional contracts used to resolve contract & contract key lookups. Optional
- domain_id
- *string*
-
- Must be a valid domain ID Required

com/daml/ledger/api/v2/completion.proto

Completion

A completion represents the status of a submitted command on the ledger: it can be successful or failed.

Field	Type	Label	Description
command_id	*string*		The ID of the succeeded or failed command. Must be a valid LedgerString (as described in `value.proto`). Required
status	google.rpc.Status		Identifies the exact type of the error. It uses the same format of conveying error details as it is used for the RPC responses of the APIs. Optional
update_id	*string*		The update_id of the transaction or reassignment that resulted from the command with command_id. Only set for successfully executed commands. Must be a valid LedgerString (as described in `value.proto`).
application_id	*string*		The application-id or user-id that was used for the submission, as described in `commands.proto`. Must be a valid ApplicationIdString (as described in `value.proto`). Optional for historic completions where this data is not available.
act_as	*string*	repeated	The set of parties on whose behalf the commands were executed. Contains the union of `party` and `act_as` from `commands.proto`. The order of the parties need not be the same as in the submission. Each element must be a valid PartyIdString (as described in `value.proto`). Optional for historic completions where this data is not available.
submission_id	*string*		The submission ID this completion refers to, as described in `commands.proto`. Must be a valid LedgerString (as described in `value.proto`). Optional
oneof deduplication_period.deduplication_offset	*string*		Specifies the start of the deduplication period by a completion stream offset (exclusive).

Must be a valid LedgerString (as described in `value.proto`).

- oneof deduplication_period.deduplication_duration
- google.protobuf.Duration
-
- Specifies the length of the deduplication period. It is measured in record time of completions.

Must be non-negative.

com/daml/ledger/api/v2/event_query_service.proto

Archived

Field	Type	Label	Description
archived_event	*com.daml.ledger.api.v1.ArchivedEvent*		Required
domain_id	*string*		Required The domain which sequenced the archival of the contract

Created

Field	Type	Label	Description
created_event	*com.daml.ledger.api.v1.CreatedEvent*		Required
domain_id	*string*		The domain which sequenced the creation of the contract Required

GetEventsByContractIdResponse

Field	Type	Label	Description
created	*Created*		The create event for the contract with the `contract_id` given in the request provided it exists and has not yet been pruned. Optional
archived	*Archived*		The archive event for the contract with the `contract_id` given in the request provided such an archive event exists and it has not yet been pruned. Optional

EventQueryService

Query events by contract id.

Note that querying by contract key is not (yet) supported, as contract keys are not supported (yet) in multi-domain scenarios.

Method name	Request type	Response type	Description
GetEvents-ByContrac-tId	*.com.daml.ledger.api.v1.GetEvents-ByContrac-tIdRequest*	*.com.daml.ledger.api.v1.GetEvents-ByContrac-tIdResponse*	Get the create and the consuming exercise event for the contract with the provided ID. No events will be returned for contracts that have been pruned because they have already been archived before the latest pruning offset.

com/daml/ledger/api/v2/package_service.proto

GetPackageRequest

Field	Type	Label	Description
pack-age_id	*string*		The ID of the requested package. Must be a valid PackageIdString (as described in `value.proto`). Required

GetPackageStatusRequest

Field	Type	Label	Description
pack-age_id	*string*		The ID of the requested package. Must be a valid PackageIdString (as described in `value.proto`). Required

ListPackagesRequest

PackageService

Allows clients to query the Daml-LF packages that are supported by the server.

Method name	Request type	Response type	Description
ListPack-ages	*ListPackagesRe-quest*	*.com.daml.ledger.api.v1.ListPack-agesResponse*	Returns the identifiers of all supported packages.
GetPackage	*GetPack-ageRequest*	*.com.daml.ledger.api.v1.GetPack-ageResponse*	Returns the contents of a single package.
GetPack-ageStatus	*GetPackageSta-tusRequest*	*.com.daml.ledger.api.v1.GetPack-ageStatusResponse*	Returns the status of a single package.

com/daml/ledger/api/v2/participant_offset.proto

ParticipantOffset

Describes a specific point on the participant. This is a participant local value: a participant offset is meaningful only in the context of its participant. Different participants may associate different offsets to the same change synchronized over a domain, and conversely, the same literal participant offset may refer to different changes on different participants.

This is also a unique index of the changes which happened on the virtual shared ledger. Participant offset define an order, which is the same in which order the updates are visible as subscribing to the `UpdateService`. This ordering is also a fully causal ordering for one specific domain: for two updates synchronized by the same domain, the one with a bigger participant offset happened after than the one with a smaller participant offset. Please note this is not true for updates synchronized by a different domain. Accordingly, the participant offset order may deviate from the order of the changes on the virtual shared ledger.

The Ledger API endpoints that take offsets allow to specify portions of the participant that are relevant for the client to read.

Offsets returned by the Ledger API can be used as-is (e.g. to keep track of processed transactions and provide a restart point to use in case of need).

The format of absolute offsets is opaque to the client: no client-side transformation of an offset is guaranteed to return a meaningful offset.

The server implementation ensures internally that offsets are lexicographically comparable.

Field	Type	Label	Description
oneof value.absolute	*string*		The format of this string is specific to the participant and opaque to the client.
oneof value.boundary	*ParticipantOffset.ParticipantBoundary*		

ParticipantOffset.ParticipantBoundary

Name	Number	Description
PARTICIPANT_BEGIN	0	Refers to the first transaction.
PARTICIPANT_END	1	Refers to the currently last transaction, which is a moving target.

com/daml/ledger/api/v2/reassignment.proto

AssignedEvent

Records that a contract has been assigned, and it can be used on the target domain.

Field	Type	Label	Description
source	*string*		The ID of the source domain. Must be a valid domain ID. Required
target	*string*		The ID of the target domain. Must be a valid domain ID. Required
unassign_id	*string*		The ID from the unassigned event. For correlation capabilities. For one contract the (unassign_id, source domain) pair is unique. Must be a valid LedgerString (as described in `value.proto`). Required
submitter	*string*		Party on whose behalf the assign command was executed. Must be a valid PartyIdString (as described in `value.proto`). Required
reassignment_counter	*uint64*		Each corresponding assigned and unassigned event has the same reassignment_counter. This strictly increases with each unassign command for the same contract. Creation of the contract corresponds to reassignment_counter equals zero. Required
created_event	*com.daml.ledger.api.v1.CreatedEvent*		Required

Reassignment

Complete view of an on-ledger reassignment.

Field	Type	Label	Description
update_id	*string*		Assigned by the server. Useful for correlating logs. Must be a valid LedgerString (as described in `value.proto`). Required
command_id	*string*		The ID of the command which resulted in this reassignment. Missing for everyone except the submitting party on the submitting participant. Must be a valid LedgerString (as described in `value.proto`). Optional
workflow_id	*string*		The workflow ID used in reassignment command submission. Only set if the `workflow_id` for the command was set. Must be a valid LedgerString (as described in `value.proto`). Optional
offset	*string*		The absolute offset. The format of this field is described in `participant_offset.proto`. Required
oneof event.unassigned_event	*UnassignedEvent*		
oneof event.assigned_event	*AssignedEvent*		

UnassignedEvent

Records that a contract has been unassigned, and it becomes unusable on the source domain

Field	Type	Label	Description
unassign_id	*string*		The ID of the unassignment. This needs to be used as an input for a assign ReassignmentCommand. For one contract the (unassign_id, source domain) pair is unique. Must be a valid LedgerString (as described in `value.proto`). Required
contract_id	*string*		The ID of the reassigned contract. Must be a valid LedgerString (as described in `value.proto`). Required
template_id	*com.daml.ledger.api.v1.Identifier*		The template of the reassigned contract. Required
source	*string*		The ID of the source domain Must be a valid domain ID Required
target	*string*		The ID of the target domain Must be a valid domain ID Required
submitter	*string*		Party on whose behalf the unassign command was executed. Must be a valid PartyIdString (as described in `value.proto`). Required
reassignment_counter	*uint64*		Each corresponding assigned and unassigned event has the same reassignment_counter. This strictly increases with each unassign command for the same contract. Creation of the contract corresponds to reassignment_counter equals zero. Required
assignment_exclusivity	*google.protobuf.Timestamp*		Assignment exclusivity Before this time (measured on the target domain), only the submitter of the unassignment can initiate the assignment Defined for reassigning participants. Optional
witness_parties	*string*	repeated	The parties that are notified of this event. Required

com/daml/ledger/api/v2/reassignment_command.proto

AssignCommand

Assign a contract

Field	Type	Label	Description
unassign_id	*string*		The ID from the unassigned event to be completed by this assignment. Must be a valid LedgerString (as described in `value.proto`). Required
source	*string*		The ID of the source domain Must be a valid domain ID Required
target	*string*		The ID of the target domain Must be a valid domain ID Required

ReassignmentCommand

Field	Type	Label	Description
workflow_id	*string*		Identifier of the on-ledger workflow that this command is a part of. Must be a valid LedgerString (as described in `value.proto`). Optional
application_id	*string*		Uniquely identifies the application or participant user that issued the command. Must be a valid ApplicationIdString (as described in `value.proto`). Required unless authentication is used with a user token or a custom token specifying an application-id. In that case, the token's user-id, respectively application-id, will be used for the request's application_id.
command_id	*string*		Uniquely identifies the command. The triple (application_id, submitter, command_id) constitutes the change ID for the intended ledger change. The change ID can be used for matching the intended ledger changes with all their completions. Must be a valid LedgerString (as described in `value.proto`). Required
submitter	*string*		Party on whose behalf the command should be executed. If ledger API authorization is enabled, then the authorization metadata must authorize the sender of the request to act on behalf of the given party. Must be a valid PartyIdString (as described in `value.proto`). Required
oneof command.unassign_command	*UnassignCommand*		
oneof command.assign_command	*AssignCommand*		
submission_id	*string*		A unique identifier to distinguish completions for different submissions with the same change ID. Typically a random UUID. Applications are expected to use a different UUID for each retry of a submission with the same change ID. Must be a valid LedgerString (as described in `value.proto`).

If omitted, the participant or the committer may set a value of their choice. Optional

UnassignCommand

Unassign a contract

Field	Type	Label	Description
contract_id	*string*		The ID of the contract the client wants to unassign. Must be a valid LedgerString (as described in `value.proto`). Required
source	*string*		The ID of the source domain Must be a valid domain ID Required
target	*string*		The ID of the target domain Must be a valid domain ID Required

com/daml/ledger/api/v2/state_service.proto

ActiveContract

Field	Type	Label	Description
created_event	*com.daml.ledger.api.v1.CreatedEvent*		Required
domain_id	*string*		A valid domain ID Required
reassignment_counter	*uint64*		Each corresponding assigned and unassigned event has the same reassignment_counter. This strictly increases with each unassign command for the same contract. Creation of the contract corresponds to reassignment_counter equals zero. This field will be the reassignment_counter of the latest observable activation event on this domain, which is before the active_at_offset. Required

GetActiveContractsRequest

Field	Type	Label	Description
filter	*TransactionFilter*		Templates to include in the served snapshot, per party. Required
verbose	*bool*		If enabled, values served over the API will contain more information than strictly necessary to interpret the data. In particular, setting the verbose flag to true triggers the ledger to include labels for record fields. Optional
active_at_offset	*string*		The offset at which the snapshot of the active contracts will be computed. Must be no greater than the current ledger end offset. Must be greater than or equal to the last pruning offset. If not set the current ledger end offset will be used. Optional

GetActiveContractsResponse

Field	Type	Label	Description
offset	*string*		Included only in the last message. The client should start consuming the transactions endpoint with this offset. The format of this field is described in `participant_offset.proto`.
workflow_id	*string*		The workflow ID used in command submission which corresponds to the contract_entry. Only set if the `workflow_id` for the command was set. Must be a valid LedgerString (as described in `value.proto`). Optional
oneof contract_entry.active_contract	*ActiveContract*		The contract is active on the given domain, meaning: there was an activation event on the given domain (created, assigned), which is not followed by a deactivation event (archived, unassigned) on the same domain, until the active_at_offset. Since activeness is defined as a per domain concept, it is possible, that a contract is active on one domain, but already archived on another. There will be one such message for each domain the contract is active on.
oneof contract_entry.incomplete_unassigned	*IncompleteUnassigned*		Included iff the unassigned event was before or at the active_at_offset, but there was no corresponding assigned event before or at the active_at_offset.
oneof contract_entry.incomplete_assigned	*IncompleteAssigned*		Important: this message is not indicating that the contract is active on the target domain! Included iff the assigned event was before or at the active_at_offset, but there was no corresponding unassigned event before or at the active_at_offset.

GetConnectedDomainsRequest

Field	Type	Label	Description
party	*string*		The party of interest Must be a valid PartyIdString (as described in `value.proto`). Required

GetConnectedDomainsResponse

Field	Type	Label	Description
connected_domains	*GetConnectedDomainsResponse.ConnectedDomain*	repeated	

GetConnectedDomainsResponse.ConnectedDomain

Field	Type	Label	Description
domain_alias	*string*		The alias of the domain Required
domain_id	*string*		The ID of the domain Required
permission	*ParticipantPermission*		The permission on the domain Required

GetLatestPrunedOffsetsRequest

Empty for now, but may contain fields in the future.

GetLatestPrunedOffsetsResponse

Field	Type	Label	Description
participant_pruned_up_to_inclusive	*ParticipantOffset*		The offset up to which the ledger has been pruned, disregarding the state of all divulged contracts pruning.
all_divulged_contracts_pruned_up_to_inclusive	*ParticipantOffset*		The offset up to which all divulged events have been pruned on the ledger. It can be at or before the `participant_pruned_up_to_inclusive` offset. For more details about all divulged events pruning, see `PruneRequest.prune_all_divulged_contracts` in `participant_pruning_service.proto`.

GetLedgerEndRequest

GetLedgerEndResponse

Field	Type	Label	Description
offset	*ParticipantOffset*		The absolute offset of the current ledger end.

IncompleteAssigned

Field	Type	Label	Description
assigned_event	*AssignedEvent*		Required

IncompleteUnassigned

Field	Type	Label	Description
created_event	*com.daml.ledger.api.v1.CreatedEvent*		Required
unassigned_event	*UnassignedEvent*		Required

ParticipantPermission

Enum indicating the permission level that the participant has for the party whose connected domains are being listed.

Name	Number	Description
Submission	0	
Confirmation	1	participant can only confirm transactions
Observation	2	participant can only observe transactions

StateService

Allows clients to get state from the ledger.

Method name	Request type	Response type	Description
GetActiveContracts	*GetActiveContractsRequest*	*GetActiveContractsResponse*	Returns a stream of the snapshot of the active contracts and incomplete reassignments at a ledger offset. If there are no active contracts, the stream returns a single response message with the offset at which the snapshot has been taken. Clients SHOULD use the offset in the last GetActiveContractsResponse message to continue streaming transactions with the update service. Clients SHOULD NOT assume that the set of active contracts they receive reflects the state at the ledger end.
GetConnectedDomains	*GetConnectedDomainsRequest*	*GetConnectedDomainsResponse*	Get the list of connected domains at the time of the query.
GetLedgerEnd	*GetLedgerEndRequest*	*GetLedgerEndResponse*	Get the current ledger end. Subscriptions started with the returned offset will serve events after this RPC was called.
GetLatestPrunedOffsets	*GetLatestPrunedOffsetsRequest*	*GetLatestPrunedOffsetsResponse*	Get the latest successfully pruned ledger offsets

com/daml/ledger/api/v2/testing/time_service.proto

GetTimeRequest

GetTimeResponse

Field	Type	Label	Description
current_time	google.protobuf.Timestamp		The current time according to the ledger server.

SetTimeRequest

Field	Type	Label	Description
current_time	google.protobuf.Timestamp		MUST precisely match the current time as it's known to the ledger server.
new_time	google.protobuf.Timestamp		The time the client wants to set on the ledger. MUST be a point int time after `current_time`.

TimeService

Optional service, exposed for testing static time scenarios.

Method name	Request type	Response type	Description
GetTime	*GetTimeRequest*	*GetTimeResponse*	Returns the current time according to the ledger server.
SetTime	*SetTimeRequest*	.google.protobuf.Empty	Allows clients to change the ledger's clock in an atomic get-and-set operation.

com/daml/ledger/api/v2/transaction.proto

Transaction

Filtered view of an on-ledger transaction's create and archive events.

Field	Type	Label	Description
update_id	*string*		Assigned by the server. Useful for correlating logs. Must be a valid LedgerString (as described in `value.proto`). Required
command_id	*string*		The ID of the command which resulted in this transaction. Missing for everyone except the submitting party. Must be a valid LedgerString (as described in `value.proto`). Optional
workflow_id	*string*		The workflow ID used in command submission. Must be a valid LedgerString (as described in `value.proto`). Optional
effective_at	google.protobuf.Timestamp		Ledger effective time. Must be a valid LedgerString (as described in `value.proto`). Required
events	*com.daml.ledger.api.v1.Event*	repeated	The collection of events. Only contains `CreatedEvent` or `ArchivedEvent`. Required
offset	*string*		The absolute offset. The format of this field is described in `participant_offset.proto`. Required
domain_id	*string*		A valid domain ID. Identifies the domain that synchronized the transaction. Required

TransactionTree

Complete view of an on-ledger transaction.

Field	Type	Label	Description
update_id	*string*		Assigned by the server. Useful for correlating logs. Must be a valid LedgerString (as described in `value.proto`). Required
command_id	*string*		The ID of the command which resulted in this transaction. Missing for everyone except the submitting party. Must be a valid LedgerString (as described in `value.proto`). Optional
workflow_id	*string*		The workflow ID used in command submission. Only set if the `workflow_id` for the command was set. Must be a valid LedgerString (as described in `value.proto`). Optional
effective_at	google.protobuf.Timestamp		Ledger effective time. Required
offset	*string*		The absolute offset. The format of this field is described in `participant_offset.proto`. Required
events_by_id	*TransactionTree.EventsByIdEntry*	repeated	Changes to the ledger that were caused by this transaction. Nodes of the transaction tree. Each key be a valid LedgerString (as describe in `value.proto`). Required
root_event_ids	*string*	repeated	Roots of the transaction tree. Each element must be a valid LedgerString (as describe in `value.proto`). The elements are in the same order as the commands in the corresponding Commands object that triggered this transaction. Required
domain_id	*string*		A valid domain ID. Identifies the domain that synchronized the transaction. Required

TransactionTree.EventsByIdEntry

Field	Type	Label	Description
key	*string*		
value	*com.daml.ledger.api.v1.TreeEvent*		

com/daml/ledger/api/v2/transaction_filter.proto

TransactionFilter

A filter both for filtering create and archive events as well as for filtering transaction trees.

Field	Type	Label	Description
filters_by_party	*TransactionFilter.FiltersByPartyEntry*	repeated	Each key must be a valid PartyIdString (as described in `value.proto`). The interpretation of the filter depends on the stream being filtered: (1) For **transaction tree streams** all party keys used as wildcard filters, and all subtrees whose root has one of the listed parties as an informee are returned. If there are InclusiveFilters, those will control returned `CreateEvent` fields were applicable, but not used for template/interface filtering. (2) For **transaction and active-contract-set streams** create and archive events are returned for all contracts whose stakeholders include at least one of the listed parties and match the per-party filter. Required

TransactionFilter.FiltersByPartyEntry

Field	Type	Label	Description
key	*string*		
value	*com.daml.ledger.api.v1.Filters*		

com/daml/ledger/api/v2/update_service.proto

GetTransactionByEventIdRequest

Field	Type	Label	Description
event_id	*string*		The ID of a particular event. Must be a valid LedgerString (as described in `value.proto`). Required
requesting_parties	*string*	repeated	The parties whose events the client expects to see. Events that are not visible for the parties in this collection will not be present in the response. Each element must be a valid PartyIdString (as described in `value.proto`). Required

GetTransactionByIdRequest

Field	Type	Label	Description
update_id	*string*		The ID of a particular transaction. Must be a valid LedgerString (as describe in `value.proto`). Required
requesting_parties	*string*	repeated	The parties whose events the client expects to see. Events that are not visible for the parties in this collection will not be present in the response. Each element be a valid PartyIdString (as describe in `value.proto`). Required

GetTransactionResponse

Field	Type	Label	Description
transaction	*Transaction*		Required

GetTransactionTreeResponse

Field	Type	Label	Description
transaction	*TransactionTree*		Required

GetUpdateTreesResponse

Field	Type	Label	Description
oneof update.transaction_tree	*TransactionTree*		
oneof update.reassignment	*Reassignment*		

GetUpdatesRequest

Field	Type	Label	Description
begin_exclusive	*ParticipantOffset*		Beginning of the requested ledger section. The response will only contain transactions whose offset is strictly greater than this. Required
end_inclusive	*ParticipantOffset*		End of the requested ledger section. The response will only contain transactions whose offset is less than or equal to this. Optional, if not set, the stream will not terminate.
filter	*TransactionFilter*		Requesting parties with template filters. Template filters must be empty for GetUpdateTrees requests. Required
verbose	*bool*		If enabled, values served over the API will contain more information than strictly necessary to interpret the data. In particular, setting the verbose flag to true triggers the ledger to include labels, record and variant type ids for record fields. Optional

GetUpdatesResponse

Field	Type	Label	Description
oneof update.transaction	*Transaction*		
oneof update.reassignment	*Reassignment*		

UpdateService

Allows clients to read updates (transactions and reassignments) from the ledger.

`GetUpdates` and `GetUpdateTrees` provide a comprehensive stream of updates/changes which happened on the virtual shared ledger. These streams are indexed with ledger offsets, which are strictly increasing. The virtual shared ledger consist of changes happening on multiple domains which are connected to the serving participant. Each update belongs to one domain, this is provided in the result (the `domain_id` field in `Transaction` and `TransactionTree` for transactions, the `source` field in `UnassignedEvent` and the `target` field in `AssignedEvent`). Consumers can rely on strong causal guarantees on the virtual shared ledger for a single domain: updates which have greater offsets are happened after than updates with smaller offsets for the same domain. Across different domains this is not guaranteed.

Method name	Request type	Response type	Description
GetUpdates	*GetUpdatesRequest*	*GetUpdatesResponse*	Read the ledger's filtered transaction stream and related reassignments for a set of parties. For transactions it lists only creates and archives, but no other events. Omits all events on transient contracts, i.e., contracts that were both created and archived in the same transaction.
GetUpdateTrees	*GetUpdatesRequest*	*GetUpdateTreesResponse*	Read the ledger's complete transaction tree stream and related reassignments for a set of parties. The stream will be filtered only by the parties as wildcard parties. The template/interface filters describe the respective fields in the `CreatedEvent` results.
GetTransactionTreeByEventId	*GetTransactionByEventIdRequest*	*GetTransactionTreeResponse*	Lookup a transaction tree by the ID of an event that appears within it. For looking up a transaction instead of a transaction tree, please see GetTransactionByEventId
GetTransactionTreeById	*GetTransactionByIdRequest*	*GetTransactionTreeResponse*	Lookup a transaction tree by its ID. For looking up a transaction instead of a transaction tree, please see GetTransactionById
GetTransactionByEventId	*GetTransactionByEventIdRequest*	*GetTransactionResponse*	Lookup a transaction by the ID of an event that appears within it.
GetTransactionById	*GetTransactionByIdRequest*	*GetTransactionResponse*	Lookup a transaction by its ID.

com/daml/ledger/api/v2/version_service.proto

GetLedgerApiVersionRequest

VersionService

Allows clients to retrieve information about the ledger API version

Method name	Request type	Response type	Description
GetLedgerApiVersion	*GetLedgerApiVersionRequest*	*.com.daml.ledger.api.v1.GetLedgerApiVersionResponse*	Read the Ledger API version

Scalar Value Types

.proto type	Notes	C++ type	Java type	Python type
double		double	double	float
float		float	float	float
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int32	int	int
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	int64	long	int/long
uint32	Uses variable-length encoding.	uint32	int	int/long
uint64	Uses variable-length encoding.	uint64	long	int/long
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32	int	int
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	int64	long	int/long
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^28.	uint32	int	int
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^56.	uint64	long	int/long
sfixed32	Always four bytes.	int32	int	int
sfixed64	Always eight bytes.	int64	long	int/long
bool		bool	boolean	boolean
string	A string must always contain UTF-8 encoded or 7-bit ASCII text.	string	String	str/unicode
bytes	May contain any arbitrary sequence of bytes.	string	ByteString	str

1.12.5.9 How Daml Types are Translated to Protobuf

This page gives an overview and reference on how Daml types and contracts are represented by the Ledger API as protobuf messages, most notably:

> in the stream of transactions from the *TransactionService*
> as payload for *CreateCommand* and *ExerciseCommand* sent to *CommandSubmissionService* and *CommandService*.

The Daml code in the examples below is written in Daml *1.1*.

Notation

The notation used on this page for the protobuf messages is the same as you get if you invoke `protoc --decode=Foo < some_payload.bin`. To illustrate the notation, here is a simple definition of the messages `Foo` and `Bar`:

```
message Foo {
  string field_with_primitive_type = 1;
  Bar field_with_message_type = 2;
}

message Bar {
  repeated int64 repeated_field_inside_bar = 1;
}
```

A particular value of `Foo` is then represented by the Ledger API in this way:

```
{ // Foo
  field_with_primitive_type: "some string"
  field_with_message_type { // Bar
    repeated_field_inside_bar: 17
    repeated_field_inside_bar: 42
    repeated_field_inside_bar: 3
  }
}
```

The name of messages is added as a comment after the opening curly brace.

Records and Primitive Types

Records or product types are translated to *Record*. Here's an example Daml record type that contains a field for each primitive type:

```
data MyProductType = MyProductType with
  intField : Int
  textField : Text
  decimalField : Decimal
  boolField : Bool
  partyField : Party
  timeField : Time
  listField : [Int]
  contractIdField : ContractId SomeTemplate
```

And here's an example of creating a value of type *MyProductType*:

```
    alice <- allocateParty "Alice"
    bob <- allocateParty "Bob"
    someCid <- submit alice do createCmd SomeTemplate with owner=alice

    let myProduct = MyProductType with
                intField = 17
                textField = "some text"
                decimalField = 17.42
                boolField = False
```

(continues on next page)

(continued from previous page)

```
                partyField = bob
                timeField = datetime 2018 May 16 0 0 0
                listField = [1,2,3]
                contractIdField = someCid
```

For this data, the respective data on the Ledger API is shown below. Note that this value would be enclosed by a particular contract containing a field of type *MyProductType*. See *Contract templates* for the translation of Daml contracts to the representation by the Ledger API.

```
{ // Record
  record_id { // Identifier
    package_id: "some-hash"
    name: "Types.MyProductType"
  }
  fields { // RecordField
    label: "intField"
    value { // Value
      int64: 17
    }
  }
  fields { // RecordField
    label: "textField"
    value { // Value
      text: "some text"
    }
  }
  fields { // RecordField
    label: "decimalField"
    value { // Value
      decimal: "17.42"
    }
  }
  fields { // RecordField
    label: "boolField"
    value { // Value
      bool: false
    }
  }
  fields { // RecordField
    label: "partyField"
    value { // Value
      party: "Bob"
    }
  }
  fields { // RecordField
    label: "timeField"
    value { // Value
      timestamp: 1526428800000000
    }
  }
  fields { // RecordField
    label: "listField"
    value { // Value
      list { // List
        elements { // Value
```

(continues on next page)

(continued from previous page)

```
          int64: 1
        }
        elements { // Value
          int64: 2
        }
        elements { // Value
          int64: 3
        }
      }
    }
  }
  fields { // RecordField
    label: "contractIdField"
    value { // Value
      contract_id: "some-contract-id"
    }
  }
}
```

Variants

Variants or sum types are types with multiple constructors. This example defines a simple variant type with two constructors:

```
data MySumType = MySumConstructor1 Int
               | MySumConstructor2 (Text, Bool)
```

The constructor `MyConstructor1` takes a single parameter of type `Integer`, whereas the constructor `MyConstructor2` takes a tuple with two fields as parameter. The snippet below shows how you can create values with either of the constructors.

```
    let mySum1 = MySumConstructor1 17
    let mySum2 = MySumConstructor2 ("it's a sum", True)
```

Similar to records, variants are also enclosed by a contract, a record, or another variant.

The snippets below shows the value of `mySum1` and `mySum2` respectively as they would be transmitted on the Ledger API within a contract.

Listing 24: mySum1

```
{ // Value
  variant { // Variant
    variant_id { // Identifier
      package_id: "some-hash"
      name: "Types.MySumType"
    }
    constructor: "MyConstructor1"
    value { // Value
      int64:  17
    }
  }
}
```

Listing 25: mySum2

```
{ // Value
  variant { // Variant
    variant_id { // Identifier
      package_id: "some-hash"
      name: "Types.MySumType"
    }
    constructor: "MyConstructor2"
    value { // Value
      record { // Record
        fields { // RecordField
          label: "sumTextField"
          value { // Value
            text: "it's a sum"
          }
        }
        fields { // RecordField
          label: "sumBoolField"
          value { // Value
            bool: true
          }
        }
      }
    }
  }
}
```

Contract Templates

Contract templates are represented as records with the same identifier as the template.

This first example template below contains only the signatory party and a simple choice to exercise:

```
data MySimpleTemplateKey =
  MySimpleTemplateKey
    with
      party: Party

template MySimpleTemplate
    with
        owner: Party
    where
        signatory owner

        key MySimpleTemplateKey owner: MySimpleTemplateKey
        maintainer key.party
```

Create a Contract

Creating contracts is done by sending a *CreateCommand* to the *CommandSubmissionService* or the *CommandService*. The message to create a *MySimpleTemplate* contract with *Alice* being the owner is shown below:

```
{ // CreateCommand
  template_id { // Identifier
    package_id: "some-hash"
    name: "Templates.MySimpleTemplate"
  }
  create_arguments { // Record
    fields { // RecordField
      label: "owner"
      value { // Value
        party: "Alice"
      }
    }
  }
}
```

Receive a Contract

Contracts are received from the *TransactionService* in the form of a *CreatedEvent*. The data contained in the event corresponds to the data that was used to create the contract.

```
{ // CreatedEvent
  event_id: "some-event-id"
  contract_id: "some-contract-id"
  template_id { // Identifier
    package_id: "some-hash"
    name: "Templates.MySimpleTemplate"
  }
  create_arguments { // Record
    fields { // RecordField
      label: "owner"
      value { // Value
        party: "Alice"
      }
    }
  }
  witness_parties: "Alice"
}
```

Exercise a Choice

A choice is exercised by sending an *ExerciseCommand*. Taking the same contract template again, exercising the choice `MyChoice` would result in a command similar to the following:

```
{ // ExerciseCommand
  template_id { // Identifier
    package_id: "some-hash"
    name: "Templates.MySimpleTemplate"
  }
  contract_id: "some-contract-id"
  choice: "MyChoice"
  choice_argument { // Value
    record { // Record
      fields { // RecordField
        label: "parameter"
        value { // Value
          int64: 42
        }
      }
    }
  }
}
```

If the template specifies a key, the *ExerciseByKeyCommand* can be used. It works in a similar way as *ExerciseCommand*, but instead of specifying the contract identifier you have to provide its key. The example above could be rewritten as follows:

```
{ // ExerciseByKeyCommand
  template_id { // Identifier
    package_id: "some-hash"
    name: "Templates.MySimpleTemplate"
  }
  contract_key { // Value
    record { // Record
      fields { // RecordField
        label: "party"
        value { // Value
          party: "Alice"
        }
      }
    }
  }
  choice: "MyChoice"
  choice_argument { // Value
    record { // Record
      fields { // RecordField
        label: "parameter"
        value { // Value
          int64: 42
        }
      }
    }
  }
}
```

1.12.5.10 How Daml Types are Translated to Daml-LF

This page shows how types in Daml are translated into Daml-LF. It should help you understand and predict the generated client interfaces, which is useful when you're building a Daml-based application that uses the Ledger API or client bindings in other languages.

For an introduction to Daml-LF, see *Daml-LF*.

Primitive Types

Built-in data types in Daml have straightforward mappings to Daml-LF.

This section only covers the serializable types, as these are what client applications can interact with via the generated Daml-LF. (Serializable types are ones whose values can exist on the ledger. Function types, `Update` and `Scenario` types and any types built up from these are excluded, and there are several other restrictions.)

Most built-in types have the same name in Daml-LF as in Daml. These are the exact mappings:

Daml primitive type	Daml-LF primitive type
`Int`	`Int64`
`Time`	`Timestamp`
`()`	`Unit`
`[]`	`List`
`Decimal`	`Decimal`
`Text`	`Text`
`Date`	`Date`
`Party`	`Party`
`Optional`	`Optional`
`ContractId`	`ContractId`

Be aware that only the Daml primitive types exported by the *Prelude* module map to the Daml-LF primitive types above. That means that, if you define your own type named `Party`, it will not translate to the Daml-LF primitive `Party`.

Tuple Types

Daml tuple type constructors take types `T1, T2, …, TN` to the type `(T1, T2, …, TN)`. These are exposed in the Daml surface language through the *Prelude* module.

The equivalent Daml-LF type constructors are `daml-prim:DA.Types:TupleN`, for each particular N (where 2 <= N <= 20). This qualified name refers to the package name (`ghc-prim`) and the module name (`GHC.Tuple`).

For example: the Daml pair type `(Int, Text)` is translated to `daml-prim:DA.Types:Tuple2 Int64 Text`.

Data Types

Daml-LF has three kinds of data declarations:

Record types, which define a collection of data
Variant or **sum** types, which define a number of alternatives
Enum, which defines simplified **sum** types without type parameters nor argument.

Data type declarations in Daml (starting with the `data` keyword) are translated to record, variant or enum types. It's sometimes not obvious what they will be translated to, so this section lists many examples of data types in Daml and their translations in Daml-LF.

Record Declarations

This section uses the syntax for Daml *records* with curly braces.

Daml declaration	Daml-LF translation
`data Foo = Foo { foo1: Int; foo2: Text }`	`record Foo □ { foo1: Int64; foo2: Text }`
`data Foo = Bar { bar1: Int; bar2: Text }`	`record Foo □ { bar1: Int64; bar2: Text }`
`data Foo = Foo { foo: Int }`	`record Foo □ { foo: Int64 }`
`data Foo = Bar { foo: Int }`	`record Foo □ { foo: Int64 }`
`data Foo = Foo {}`	`record Foo □ {}`
`data Foo = Bar {}`	`record Foo □ {}`

Variant Declarations

Daml declaration	Daml-LF translation
`data Foo = Bar Int | Baz Text`	`variant Foo □ Bar Int64 | Baz Text`
`data Foo a = Bar a | Baz Text`	`variant Foo a □ Bar a | Baz Text`
`data Foo = Bar Unit | Baz Text`	`variant Foo □ Bar Unit | Baz Text`
`data Foo = Bar Unit | Baz`	`variant Foo □ Bar Unit | Baz Unit`
`data Foo a = Bar | Baz`	`variant Foo a □ Bar Unit | Baz Unit`
`data Foo = Foo Int`	`variant Foo □ Foo Int64`
`data Foo = Bar Int`	`variant Foo □ Bar Int64`
`data Foo = Foo ()`	`variant Foo □ Foo Unit`
`data Foo = Bar ()`	`variant Foo □ Bar Unit`
`data Foo = Bar { bar: Int } | Baz Text`	`variant Foo □ Bar Foo.Bar | Baz Text, record Foo.Bar □ { bar: Int64 }`
`data Foo = Foo { foo: Int } | Baz Text`	`variant Foo □ Foo Foo.Foo | Baz Text, record Foo.Foo □ { foo: Int64 }`
`data Foo = Bar { bar1: Int; bar2: Decimal } | Baz Text`	`variant Foo □ Bar Foo.Bar | Baz Text, record Foo.Bar □ { bar1: Int64; bar2: Decimal }`
`data Foo = Bar { bar1: Int; bar2: Decimal } | Baz { baz1: Text; baz2: Date }`	`data Foo □ Bar Foo.Bar | Baz Foo.Baz, record Foo.Bar □ { bar1: Int64; bar2: Decimal }, record Foo.Baz □ { baz1: Text; baz2: Date }`

Enum Declarations

Daml declaration	Daml-LF declaration
`data Foo = Bar | Baz`	`enum Foo □ Bar | Baz`
`data Color = Red | Green | Blue`	`enum Color □ Red | Green | Blue`

Banned Declarations

There are two gotchas to be aware of: things you might expect to be able to do in Daml that you can't because of Daml-LF.

The first: a single constructor data type must be made unambiguous as to whether it is a record or a variant type. Concretely, the data type declaration `data Foo = Foo` causes a compile-time error, because it is unclear whether it is declaring a record or a variant type.

To fix this, you must make the distinction explicitly. Write `data Foo = Foo {}` to declare a record type with no fields, or `data Foo = Foo ()` for a variant with a single constructor taking unit argument.

The second gotcha is that a constructor in a data type declaration can have at most one unlabelled argument type. This restriction is so that we can provide a straight-forward encoding of Daml-LF

types in a variety of client languages.

Banned declaration	Workaround
`data Foo = Foo`	`data Foo = Foo {}` to produce `record Foo □ {}` OR `data Foo = Foo ()` to produce `variant Foo □ Foo Unit`
`data Foo = Bar`	`data Foo = Bar {}` to produce `record Foo □ {}` OR `data Foo = Bar ()` to produce `variant Foo □ Bar Unit`
`data Foo = Foo Int Text`	Name constructor arguments using a record declaration, for example `data Foo = Foo { x: Int; y: Text }`
`data Foo = Bar Int Text`	Name constructor arguments using a record declaration, for example `data Foo = Bar { x: Int; y: Text }`
`data Foo = Bar | Baz Int Text`	Name arguments to the Baz constructor, for example `data Foo = Bar | Baz { x: Int; y: Text }`

Type Synonyms

Type synonyms (starting with the `type` keyword) are eliminated during conversion to Daml-LF. The body of the type synonym is inlined for all occurrences of the type synonym name.

For example, consider the following Daml type declarations.

```
type Username = Text
data User = User { name: Username }
```

The `Username` type is eliminated in the Daml-LF translation, as follows:

```
record User □ { name: Text }
```

Template Types

A *template declaration* in Daml results in one or more data type declarations behind the scenes. These data types, detailed in this section, are not written explicitly in the Daml program but are created by the compiler.

They are translated to Daml-LF using the same rules as for record declarations above.

These declarations are all at the top level of the module in which the template is defined.

Template Data Types

Every contract template defines a record type for the parameters of the contract. For example, the template declaration:

```
template Iou
  with
    issuer: Party
    owner: Party
    currency: Text
    amount: Decimal
  where
```

results in this record declaration:

```
data Iou = Iou { issuer: Party; owner: Party; currency: Text; amount: Decimal }
```

This translates to the Daml-LF record declaration:

```
record Iou □ { issuer: Party; owner: Party; currency: Text; amount: Decimal }
```

Choice Data Types

Every choice within a contract template results in a record type for the parameters of that choice. For example, let's suppose the earlier `Iou` template has the following choices:

```
    nonconsuming choice DoNothing: ()
      controller owner
      do
        return ()

    choice Transfer: ContractId Iou
      with newOwner: Party
      controller owner
      do
        updateOwner newOwner
```

This results in these two record types:

```
data DoNothing = DoNothing {}
data Transfer = Transfer { newOwner: Party }
```

Whether the choice is consuming or nonconsuming is irrelevant to the data type declaration. The data type is a record even if there are no fields.

These translate to the Daml-LF record declarations:

```
record DoNothing □ {}
record Transfer □ { newOwner: Party }
```

Names with Special Characters

All names in Daml—of types, templates, choices, fields, and variant data constructors—are translated to the more restrictive rules of Daml-LF. ASCII letters, digits, and _ underscore are unchanged in Daml-LF; all other characters must be mangled in some way, as follows:

`$` changes to `$$`,
Unicode codepoints less than 65536 translate to `$uABCD`, where `ABCD` are exactly four (zero-padded) hexadecimal digits of the codepoint in question, using only lowercase `a-f`, and Unicode codepoints greater translate to `$UABCD1234`, where `ABCD1234` are exactly eight (zero-padded) hexadecimal digits of the codepoint in question, with the same `a-f` rule.

Daml name	Daml-LF identifier
`Foo_bar`	`Foo_bar`
`baz'`	`baz$u0027`
`:+:`	`$u003a$u002b$u003a`
`naïveté`	`na$u00efvet$u00e9`
`:🙂:`	`$u003a$U0001f642$u003a`

1.12.5.11 Create Your Own Bindings

This page gets you started with creating custom bindings for a Daml Ledger.

Bindings for a language consist of two main components:

Ledger API Client stubs for the programming language, - the remote API that allows sending ledger commands and receiving ledger transactions. You have to generate **Ledger API** from the gRPC protobuf definitions in the daml repository on GitHub. **Ledger API** is documented on this page: *Use the Ledger API With gRPC*. The gRPC tutorial explains how to generate client stubs .

Codegen A code generator is a program that generates classes representing Daml contract templates in the language. These classes incorporate all boilerplate code for constructing: *CreateCommand* and *ExerciseCommand* corresponding for each Daml contract template.

Technically codegen is optional. You can construct the commands manually from the auto-generated **Ledger API** classes. However, it is very tedious and error-prone. If you are creating *ad hoc* bindings for a project with a few contract templates, writing a proper codegen may be overkill. On the other hand, if you have hundreds of contract templates in your project or are planning to build language bindings that you will share across multiple projects, we recommend including a codegen in your bindings. It will save you and your users time in the long run.

Note that for different reasons we chose codegen, but that is not the only option. There is really a broad category of metaprogramming features that can solve this problem just as well or even better than codegen; they are language-specific, but often much easier to maintain (i.e. no need to add a build step). Some examples are:

F# Type Providers
Template Haskell

Build Ledger Commands

No matter what approach you take, either manually building commands or writing a codegen to do this, you need to understand how ledger commands are structured. This section demonstrates how to build create and exercise commands manually and how it can be done using contract classes.

Create Command

Let's recall an **IOU** example from the *Quickstart guide*, where *Iou* template is defined like this:

```
template Iou
  with
    issuer : Party
    owner : Party
    currency : Text
    amount : Decimal
    observers : [Party]
```

If you do not specify any of the above fields or type their names or values incorrectly, or do not order them exactly as they are in the Daml template, the above code will compile but fail at run-time because you did not structure your create command correctly.

Exercise Command

To build *ExerciseCommand* for *Iou_Transfer*:

```
    choice Iou_Transfer : ContractId IouTransfer
      with
        newOwner : Party
      controller owner
      do create IouTransfer with iou = this; newOwner
```

Summary

When creating custom bindings for Daml Ledgers, you will need to:

generate **Ledger API** from the gRPC definitions
decide whether to write a codegen to generate ledger commands or manually build them for all contracts defined in your Daml model.

The above examples should help you get started. If you are creating custom binding or have any questions, see the *Getting Help* page for how to get in touch with us.

Links

gRPC documentation: https://grpc.io/docs/
Documentation for Protobuf well known types : https://developers.google.com/protocol-buffers/docs/reference/google.protobuf
Daml Ledger API gRPC Protobuf definitions
- current main: https://github.com/digital-asset/daml/tree/main/ledger-api/grpc-definitions
- for specific versions: https://github.com/digital-asset/daml/releases

Required gRPC Protobuf definitions:
- https://raw.githubusercontent.com/grpc/grpc/v1.18.0/src/proto/grpc/status/status.proto
- https://raw.githubusercontent.com/grpc/grpc/v1.18.0/src/proto/grpc/health/v1/health.proto

1.12.6 Daml Off-Ledger Automation

1.12.6.1 Write Off-Ledger Automation Using Daml

The Daml smart contract language is mostly meant to provide a way to define on-ledger logic, i.e. code that defines how a transaction happens on ledger. Daml is not meant to be used as a general purpose language that can interact with your file system or network; instead, the templates and choices defined with Daml are available to be used by off-ledger logic that interacts with the ledger API. Usually this off-ledger logic is written in a general-purpose language like Java or JavaScript and the codegen allows to interact with models defined in Daml without boilerplate.

However, there are times when it would be nice to write your off-ledger logic in Daml. For relatively simple automations that don't require full access to your system's capabilities, using Daml means that you don't have to map from your on-ledger Daml types and their representation on a separate off-ledger general purpose language (either through the codegen or by manipulating the Protobuf representation of Daml types directly).

There are two tools that allow you to use Daml as an off-ledger language:

Daml Script allows you to write automations that can be triggered by any off-ledger condition, such as the availability of a file in a folder, a message coming from a broker or a user interacting with the system directly.
Daml Triggers allow a similar approach but triggered by on-ledger events, such as the creation of a contract.

In their interactions with a traditional database system Daml Scripts and Daml Triggers are analogous to SQL scripts and SQL triggers.

1.12.6.2 Daml Script

Daml Script provides a simple way of testing Daml models and getting quick feedback in Daml studio. In addition to running it in a virtual ledger in *Daml Studio*, you can also point it against an actual ledger. This means that you can use it for application scripting, to test automation logic and also for *ledger initialization*.

You can also use Daml Script interactively using *Daml REPL*.

Hint: Remember that you can access all the example code by running `daml new script-example --template script-example`

Usage

Our example for this tutorial consists of 2 templates.

First, we have a template called `Coin`:

```
template Coin
  with
    issuer : Party
    owner : Party
  where
    signatory issuer, owner
```

This template represents a coin issued to `owner` by `issuer`. `Coin` has both the `owner` and the `issuer` as signatories.

Second, we have a template called `CoinProposal`:

```
template CoinProposal
  with
    coin : Coin
  where
    signatory coin.issuer
    observer coin.owner

    choice Accept : ContractId Coin
      controller coin.owner
      do create coin
```

`CoinProposal` is only signed by the `issuer` and it provides a single `Accept` choice which, when exercised by the controller will create the corresponding `Coin`.

Having defined the templates, we can now move on to write Daml scripts that operate on these templates. To get access to the API used to implement Daml scripts, you need to add the `daml-script` library to the `dependencies` field in `daml.yaml`.

```
dependencies:
  - daml-prim
  - daml-stdlib
  - daml-script
```

We also enable the `ApplicativeDo` extension. We will see below why this is useful.

```
{-# LANGUAGE ApplicativeDo #-}

module ScriptExample where

import DA.Time
import Daml.Script
```

Since on an actual ledger parties cannot be arbitrary strings, we define a record containing all the parties that we will use in our script so that we can easily swap them out.

```
data LedgerParties = LedgerParties with
  bank : Party
  alice : Party
  bob : Party
```

Let us now write a function to initialize the ledger with 3 `CoinProposal` contracts and accept 2 of them. This function takes the `LedgerParties` as an argument and returns a value of type `Script ()` which is Daml script's equivalent of `Scenario ()`.

```
initialize : LedgerParties -> Script ()
initialize parties = do
```

First we create the proposals. To do so, we use the `submit` function to submit a transaction. The first argument is the party submitting the transaction. In our case, we want all proposals to be created by the bank so we use `parties.bank`. The second argument must be of type `Commands a` so in our case `Commands (ContractId CoinProposal, ContractId CoinProposal, ContractId CoinProposal)` corresponding to the 3 proposals that we create. However, `Commands` requires that the individual commands do not depend on each other. This matches the restriction on the Ledger API where a transaction consists of a list of commands. Using `ApplicativeDo` we can still use `do`-notation as long as we respect this and the last statement in the `do`-block is of the form `return expr` or `pure expr`. In `Commands` we use `createCmd` instead of `create` and `exerciseCmd` instead of `exercise`.

```
  (coinProposalAlice, coinProposalBob, coinProposalBank) <- submit parties.bank $
↪do
    coinProposalAlice <- createCmd (CoinProposal (Coin parties.bank parties.
↪alice))
    coinProposalBob <- createCmd (CoinProposal (Coin parties.bank parties.bob))
    coinProposalBank <- createCmd (CoinProposal (Coin parties.bank parties.bank))
    pure (coinProposalAlice, coinProposalBob, coinProposalBank)
```

Now that we have created the `CoinProposal`s, we want `Alice` and `Bob` to accept the proposal while the `Bank` will ignore the proposal that it has created for itself. To do so we use separate `submit` statements for `Alice` and `Bob` and call `exerciseCmd`.

```
  coinAlice <- submit parties.alice $ exerciseCmd coinProposalAlice Accept
  coinBob <- submit parties.bob $ exerciseCmd coinProposalBob Accept
```

Finally, we call `pure ()` on the last line of our script to match the type `Script ()`.

```
  pure ()
```

Party Management

We have now defined a way to initialize the ledger so we can write a test that checks that the contracts that we expect exist afterwards.

First, we define the signature of our test. We will create the parties used here in the test, so it does not take any arguments.

```
test : Script ()
test = do
```

Now, we create the parties using the `allocateParty` function. This uses the party management service to create new parties with the given display name. Note that the display name does not identify a party uniquely. If you call `allocateParty` twice with the same display name, it will create 2 different parties. This is very convenient for testing since a new party cannot see any old contracts on the ledger so using new parties for each test removes the need to reset the ledger. We factor out party allocation into a function so we can reuse it in later sections.

```
allocateParties : Script LedgerParties
allocateParties = do
  alice <- allocateParty "alice"
  bob <- allocateParty "bob"
  bank <- allocateParty "Bank"
  pure (LedgerParties bank alice bob)
```

We now call the `initialize` function that we defined before on the parties that we have just allocated.

```
  initialize parties
```

Queries

To verify the contracts on the ledger, we use the `query` function. We pass it the type of the template and a party. It will then give us all active contracts of the given type visible to the party. In our example, we expect to see one active `CoinProposal` for `bank` and one `Coin` contract for each of `Alice` and `Bob`. We get back list of `(ContractId t, t)` pairs from `query`. In our tests, we do not need the contract ids, so we throw them away using `map snd`.

```
  proposals <- query @CoinProposal bank
  assertEq [CoinProposal (Coin bank bank)] (map snd proposals)

  aliceCoins <- query @Coin alice
  assertEq [Coin bank alice] (map snd aliceCoins)

  bobCoins <- query @Coin bob
  assertEq [Coin bank bob] (map snd bobCoins)
```

Interfaces

To use interfaces within Daml code, the target language version must be at least `1.15`.

```
build-options:
  - --target=1.15
```

Now we can define an `Asset` interface which can be implemented by the `Coin` template. We also define `AssetInfo` for use as the viewtype.

```
data AssetInfo = AssetInfo { info : Text } deriving (Eq,Show)

interface Asset where
  viewtype AssetInfo

  interface instance Asset for Coin where
    view = AssetInfo { info = "A Coin" }
```

Now we use the `queryInterface` function. We pass it the type of the interface and a party. It will return a list of active contract views for the given interface type. As before we throw away the contract ids using `map snd`.

```
  aliceAssets <- queryInterface @Asset alice
  assertEq [Some $ AssetInfo "A Coin"] (map snd aliceAssets)
```

Run a Script

To run our script, we first build it with `daml build` and then run it by pointing to the DAR, the name of our script, and the host and port our ledger is running on.

```
daml script --dar .daml/dist/script-example-0.0.1.dar --script-name Scrip-
tExample:test --ledger-host localhost --ledger-port 6865
```

Up to now, we have worked with a script (`test`) that is entirely self-contained. This is fine for running unit-test type script in the IDE, but for more complex use-cases you may want to vary the inputs of a script and inspect its outputs, ideally without having to recompile it. To that end, the `daml script` command supports the flags `--input-file` and `--output-file`. Both flags take a filename, and said file will be read/written as JSON, following the *Daml-LF JSON Encoding*.

The `--output-file` option instructs `daml script` to write the result of the given `--script-name` to the given filename (creating the file if it does not exist; overwriting it otherwise). This is most useful if the given program has a type `Script b`, where `b` is a meaningful value. In our example, we can use this to write out the party ids that have been allocated by `allocateParties`:

```
daml script --dar .daml/dist/script-example-0.0.1.dar --script-name
ScriptExample:allocateParties --ledger-host localhost --ledger-port 6865
--output-file ledger-parties.json
```

The resulting file will look similar to the following but the actual party IDs will be different each time you run it:

```
{
  "bank": "party-93affbfe-8717-4996-990c-
↪9f4c5a889663::12201d00faa0968d7ab81e63ad6ad4ee0d31b08a3581b1d8596e68a1356f27519ccb
↪",
```

(continues on next page)

(continued from previous page)

```
  "alice": "party-99595f45-75e3-4373-997c-
↪fbdf899439f7::12201d00faa0968d7ab81e63ad6ad4ee0d31b08a3581b1d8596e68a1356f27519ccb
↪",
  "bob": "party-6e38e1ed-c070-4ded-ba20-
↪073e0dbdb13c::12201d00faa0968d7ab81e63ad6ad4ee0d31b08a3581b1d8596e68a1356f27519ccb
↪"
}
```

Next, we want to call the `initialize` function with those parties using the `--input-file` flag. If the `--input-file` flag is specified, the `--script-name` flag must point to a function of one argument returning a `Script`, and the function will be called with the result of parsing the input file as its argument. For example, we can initialize our ledger using the `initialize` function defined above.

Using the previously created `-ledger-parties.json` file, we can initialize our ledger as follows:

```
daml script --dar .daml/dist/script-example-0.0.1.dar --script-name Script-
tExample:initialize --ledger-host localhost --ledger-port 6865 --input-file
ledger-parties.json
```

Use Daml Script for Ledger Initialization

You can use Daml script to initialize a ledger on startup. To do so, specify an `init-script: Script-tExample:initializeUser` field in your `daml.yaml`. This will automatically be picked up by `daml start` and used to initialize sandbox. During development not being able to control party ids can often be inconvenient. Here, we rely on *users* which do put us in control of their id. User ids can be used in Navigator, triggers & other tools instead of party ids.

```
initializeUser : Script ()
initializeUser = do
  parties <- allocateParties
  bank <- validateUserId "bank"
  alice <- validateUserId "alice"
  bob <- validateUserId "bob"
  _ <- createUser (User bank (Some parties.bank)) [CanActAs parties.bank]
  _ <- createUser (User alice (Some parties.alice)) [CanActAs parties.alice]
  _ <- createUser (User bob (Some parties.bob)) [CanActAs parties.bob]
  initialize parties
```

Migrate From Scenarios

Existing scenarios that you used for ledger initialization can be translated to Daml script but there are a few things to keep in mind:

1. You need to add `daml-script` to the list of dependencies in your `daml.yaml`.
2. You need to import the `Daml.Script` module.
3. Calls to `create`, `exercise`, `exerciseByKey` and `createAndExercise` need to be suffixed with `Cmd`, e.g., `createCmd`.
4. Instead of specifying a `scenario` field in your `daml.yaml`, you need to specify an `init-script` field. The initialization script is specified via `Module:identifier` for both fields.

5. In Daml script, `submit` and `submitMustFail` are limited to the functionality provided by the ledger API: A list of independent commands consisting of `createCmd`, `exerciseCmd`, `createAndExerciseCmd` and `exerciseByKeyCmd`. There are two issues you might run into when migrating an existing scenario:
   1. Your commands depend on each other, e.g., you use the result of a `create` within a following command in the same `submit`. In this case, you have two options: If it is not important that they are part of a single transaction, split them into multiple calls to `submit`. If you do need them to be within the same transaction, you can move the logic to a choice and call that using `createAndExerciseCmd`.
   2. You use something that is not part of the 4 ledger API command types, e.g., `fetch`. For `fetch` and `fetchByKey`, you can instead use `queryContractId` and `queryContractKey` with the caveat that they do not run within the same transaction. Other types of `Update` statements can be moved to a choice that you call via `createAndExerciseCmd`.
6. Instead of Scenario's `getParty`, Daml Script provides you with `allocateParty` and `allocatePartyWithHint`. There are a few important differences:
   1. Allocating a party always gives you back a new party (or fails). If you have multiple calls to `getParty` with the same string and expect to get back the same party, you should instead allocate the party once at the beginning and pass it along to the rest of the code.
   2. If you want to allocate a party with a specific party id, you can use `allocatePartyWithHint x (PartyIdHint x)` as a replacement for *getParty* x. Note that while this is supported in Daml Studio, some ledgers can behave differently and ignore the party id hint or interpret it another way. Try to not rely on any specific party id.
7. Instead of `pass` and `passToDate`, Daml Script provides `passTime` and `setTime`.

Use Daml Script with the IDE Ledger

Similarly to running `daml test` or when running a script in VSCode itself via the provided buttons, you can use `daml script` to run the scripts in a given DAR file within the IDE Ledger. This is a fully in-memory child process of `daml script`, allowing you to quickly invoke a script without having to spin up a ledger in the background.

To run `daml script` in this mode, you should provide the `--ide-ledger` flag. This flag is not compatible with `--ledger-host`, `--ledger-port`, `--participant-config` (described more in the next section), and `--json-api`. Note that since this uses an in-memory ledger, no state will be preserved once the script finishes. You will only receive a success flag and, optionally, the script result if you use `--output-file`.

Use Daml Script in Canton

So far, we have run Daml script against a single participant node. It is also possible to run it in a setting where different parties are hosted on different participant nodes. To do so, pass the `--participant-config participant-config.json` file to `daml script` instead of `--ledger-host` and `ledger-port`. You can generate this file by calling *utils.generate_daml_script_participants_conf(defaultParticipant = Some(one))* in the canton console or in *the bootstrap scripts*.

The generated file will look similar to the one shown below:

```
{
    "default_participant": {"host": "localhost", "port": 6866},
```

(continues on next page)

(continued from previous page)

```
    "participants": {
        "one": {"host": "localhost", "port": 6866},
        "two": {"host": "localhost", "port": 6865}
    },
    "party_participants": {"alice": "one", "bob": "two"}
}
```

This will define a participant called `one`, declare `one` as the default participant and it defines that the party `alice` is hosted on participant `one`. Whenever you submit something as party, we will use the participant for that party or if none is specified `default_participant`.

If you use `utils.generate_daml_script_participants_conf()` without a default participant, the *default_participant* won't be defined and therefore using a party with an unspecified participant is an error.

`allocateParty` will also use the `default_participant`. If you want to allocate a party on a specific participant, you can use `allocatePartyOn` which accepts the participant name as an extra argument.

Hints for synchronizing contracts on multiple-participant Canton

When you create a contract on `participant1` and try to use it on `participant2`, you can run into synchronization issues where `participant2` doesn't see the contract yet. One option to workaround this limitation is to poll until the contract is visible. In the example below, the `bank` and `alice` parties are allocated on two different participants and to avoid synchronization issues, we wait until the contract is visible on `alice` participant.

```
tries : Int
tries = 60

waitForCid : (Template t, HasAgreement t) => Int -> Party -> ContractId t ->
↪Script ()
waitForCid tries p cid
  | tries <= 0 = abort $ "Cid " <> show cid <> " did not appear"
  | otherwise = do
    r <- queryContractId p cid
    case r of
      None -> do
        sleep delay
        waitForCid (tries - 1) p cid
      Some _ -> pure ()
  where delay = seconds 1

testWithSync: LedgerParties -> Script ()
testWithSync parties = do
  coinProposalAlice <- submit parties.bank $ createCmd (CoinProposal (Coin
↪parties.bank parties.alice))
  waitForCid tries parties.alice coinProposalAlice
  coinAlice <- submit parties.alice $ exerciseCmd coinProposalAlice Accept
  pure ()
```

Run Daml Script Against Ledgers with Authorization

To run Daml Script against a ledger that verifies authorization, you need to specify an access token. There are two ways of doing that:

1. Specify a single access token via `--access-token-file path/to/jwt`. This token will then be used for all requests so it must provide claims for all parties that you use in your script.
2. If you need multiple tokens, e.g., because you only have single-party tokens you can define the `access_token` field in the participant config specified via `--participant-config`. Note that you can specify the same participant twice if you want different auth tokens. The file should be of the format

```
{
    "default_participant": {"host": "localhost", "port": 6866, "access_token":
↪"default_jwt", "application_id": "myapp"},
    "participants": {
        "one": {"host": "localhost", "port": 6866, "access_token": "jwt_for_alice
↪", "application_id": "myapp"},
        "two": {"host": "localhost", "port": 6865, "access_token": "jwt_for_bob",
↪"application_id": "myapp"}
    },
    "party_participants": {"alice": "one", "bob": "two"}
}
```

If you specify both `--access-token-file` and `--participant-config`, the participant config takes precedence and the token from the file will be used for any participant that does not have a token specified in the config.

Run Daml Script Against the HTTP JSON API

In some cases, you only have access to the *HTTP JSON API* but not to the gRPC of a ledger, e.g., on Daml Hub. For this use case, Daml script can be run against the JSON API. Note that if you do have access to the gRPC Ledger API, running Daml script against the JSON API does not have any advantages.

To run Daml script against the JSON API you have to pass the `--json-api` parameter to `daml script`. There are a few differences and limitations compared to running Daml Script against the gRPC Ledger API:

1. When running against the JSON API, the `--host` argument has to contain an `http://` or `https://` prefix, e.g., `daml script --host http://localhost --port 7575 --json-api`.
2. The JSON API only supports single-command submissions. This means that within a single call to `submit` you can only execute one ledger API command, e.g., one `createCmd` or one `exerciseCmd`.
3. The JSON API requires authorization tokens even when it is run against a ledger that doesn't verify authorization. The section on *authorization* describes how to specify the tokens.
4. The parties used for command submissions and queries must match the parties specified in the token exactly. For command submissions that means `actAs` and `readAs` must match exactly what you specified whereas for queries the union of `actAs` and `readAs` must match the parties specified in the query.
5. If you use multiple parties within your Daml Script, you need to specify one token per party or every submission and query must specify all parties of the multi-party token.

6. `getTime` will always return the Unix epoch in static time mode since the time service is not exposed via the JSON API.
7. `setTime` is not supported and will throw a runtime error.

1.12.6.3 Daml Triggers - Off-Ledger Automation in Daml

In addition to the actual Daml logic which is uploaded to the Ledger and the UI, Daml applications often need to automate certain interactions with the ledger. This is commonly done in the form of a ledger client that listens to the transaction stream of the ledger and when certain conditions are met, e.g., when a template of a given type has been created, the client sends commands to the ledger to create a template of another type.

It is possible to write these clients in a language of your choice, such as JavaScript, using the HTTP JSON API. However, that introduces an additional layer of friction: you now need to translate between the template and choice types in Daml and a representation of those Daml types in the language you are using for your client. Daml triggers address this problem by allowing you to write certain kinds of automation directly in Daml, reusing all the Daml types and logic that you have already defined. Note that, while the logic for Daml triggers is written in Daml, they act like any other ledger client: they are executed separately from the ledger, they do not need to be uploaded to the ledger and they do not allow you to do anything that any other ledger client could not do.

If you don't want to follow along, but still want to get the final code for this section to play with, you can get it by running:

```
daml new --template=gsg-trigger gsg-trigger
```

How To Think About Triggers

It is tempting to think of Daml Triggers as snippets of code that react to ledger events . However, this is not the best way to think about them; while it will work in some cases, in many corner cases that line of thought will lead to subtle errors.

Instead, you should think of, and write, your triggers from the perspective of correcting the current ACS to match some predefined expectations. Trigger rules should be a combination of checking those expectations on the current ACS and applying corrective actions to bring back the ACS in line with its expected state.

The trigger part is best thought of as an optimization: rather than check the ACS constantly, we only apply our rules when something happens that we believe *may* lead to the state of the ledger diverging from our expectations.

Sample Trigger

Our example for this tutorial builds upon the Getting Started Guide, specifically picking up right after the *Your First Feature* section.

We assume that our requirements are to build a chatbot that responds to every message with:

> Please, tell me more about that.

That should fool anyone and pass the Turing test, easily.

As explained above, while the layman description may be responds to every message , our technical description is better phrased as ensure that, at all times, the last message we can see has been sent by us; if that is not the case, the corrective action is to send a response to the last message we can see .

Daml Trigger Basics

A Daml trigger is a regular Daml project that you can build using `daml build`. To get access to the API used to build a trigger, you need to add the `daml-trigger` library to the `dependencies` field in `daml.yaml`:

```
dependencies:
- daml-prim
- daml-stdlib
- daml-script
- daml-trigger
```

Note: In the specific case of the Getting Started Guide, this is already included as part of the `create-daml-app` template.

In addition to that you also need to import the `Daml.Trigger` module in your own code.

Daml triggers automatically track the active contract set (ACS), i.e., the set of contracts that have been created and have not been archived, and the commands in flight for you. In addition to that, they allow you to have user-defined state that is updated based on new transactions and command completions. For our chatbot trigger, the ACS is sufficient, so we will simply use `()` as the type of the user defined state.

To create a trigger you need to define a value of type `Trigger s` where `s` is the type of your user-defined state:

```
data Trigger s = Trigger
  { initialize : TriggerInitializeA s
  , updateState : Message -> TriggerUpdateA s ()
  , rule : Party -> TriggerA s ()
  , registeredTemplates : RegisteredTemplates
  , heartbeat : Optional RelTime
  }
```

To clarify, this is the definition in the `Daml.Trigger` library, reproduced here for illustration purposes. This is not something you need to add to your own code.

The `initialize` function is called on startup and allows you to initialize your user-defined state based on querying the active contract set.

The `updateState` function is called on new transactions and command completions and can be used to update your user-defined state based on the ACS and the transaction or completion. Since our Daml trigger does not have any interesting user-defined state, we will not go into details here.

The `rule` function is the core of a Daml trigger. It defines which commands need to be sent to the ledger based on the party the trigger is executed at, the current state of the ACS, and the user defined state. The type `TriggerA` allows you to emit commands that are then sent to the ledger, query the ACS with `query`, update the user-defined state, as well as retrieve the commands in flight with `getCommandsInFlight`. Like `Scenario` or `Update`, you can use `do` notation and `getTime` with `TriggerA`.

We can specify the templates and interfaces that our trigger will operate on. In our case, we will simply specify `AllInDar` which means that the trigger will receive events for all template and interface types defined in the DAR.

It is also possible to specify an explicit list of templates and interfaces. For example, to only receive events for the `Message` template, one would write:

```
...
registeredTemplates = RegisteredTemplates [registeredTemplate @Message],
...
```

This is mainly useful for performance reasons if your DAR contains many templates and interfaces that are not relevant for your trigger. Note that providing an explicit list of templates and interfaces also filters the result of querying the ACS using the Trigger API: contracts of the excluded templates and interfaces cannot be queried.

Note: In these examples we used templates. Note that interfaces can be passed as well wherever a template is passed, using the same *RegisteredTemplates* type. You are free to pass multiple templates and interfaces and possibly mix the two freely in a single request.

Finally, you can specify an optional heartbeat interval at which the trigger will be sent a `MHeartbeat` message. This is useful if you want to ensure that the trigger is executed at a certain rate to issue timed commands. We will not be using heartbeats in this example.

Run a No-Op Trigger

To implement a no-op trigger, one could write the following in a separate `daml/ChatBot.daml` file:

```
module NoOp where

import qualified Daml.Trigger as T

noOp : T.Trigger ()
noOp = T.Trigger with
  initialize = pure ()
  updateState = \_ -> pure ()
  rule = \_ -> do
    debug "triggered"
    pure ()
  registeredTemplates = T.AllInDar
  heartbeat = None
```

In the context of the Getting Started app, if you write the above file, then run `daml start` and `npm start` as usual, and then set up the trigger with:

```
daml trigger --dar .daml/dist/gsg-trigger-0.1.0.dar \
             --trigger-name NoOp:noOp \
             --ledger-host localhost \
             --ledger-port 6865 \
             --ledger-user "bob"
```

and then play with the app as `alice` and `bob` just like you did for *Your First Feature*, you should see the trigger command printing a line for each interaction, containing the message `triggered` as well as

other debug information.

Diversion: Updating `Message`

Before we can make our Trigger more useful, we need to think a bit more about what it is supposed to do. For example, we don't want to respond to `bob`'s own messages. We also do not want to send messages when we have not received any.

In order to start with something reasonably simple, we're going to set the rule as

> *if* the last message we can see was *not* sent by `bob`, *then* we'll send `"Please, tell me more about that."` to whoever sent the last message we can see.

This raises the question of how we can determine which message is the last one, given the current structure of a message. In order to solve that, we need to add a `Time` field to `Message`, which can be done by editing the `Message` template in `daml/User.daml` to look like:

```
template Message with
    sender: Party
    receiver: Party
    content: Text
    receivedAt: Time
  where
    signatory sender, receiver
```

This should result in Daml Studio reporting an error in the `SendMessage` choice, as it now needs to set the `receivedAt` field. Here is the updated code for `SendMessage`:

```
    -- New definition for SendMessage
    nonconsuming choice SendMessage: ContractId Message with
        sender: Party
        content: Text
      controller sender
      do
        assertMsg "Designated user must follow you back to send a message" (elem
↪sender following)
        now <- getTime
        create Message with sender, receiver = username, content, receivedAt = now
```

The `getTime` action (doc) returns the time at which the command was received by the sandbox. In more sensitive applications, this may not be sufficiently reliable, as transactions may be processed in parallel (so received at timestamp order may not match actual transaction order), and in distributed cases dishonest participants may fudge this value. It's good enough for this example, though.

Now that we have a field to sort on, and thus a way to identify the *latest* message, we can turn our attention back to our trigger code.

AutoReply

Open up the trigger code again (`daml/ChatBot.daml`), and change it to:

```
module ChatBot where

import qualified Daml.Trigger as T
import qualified User
import qualified DA.List.Total as List
import DA.Action (when)
import DA.Optional (whenSome)

autoReply : T.Trigger ()
autoReply = T.Trigger
  { initialize = pure ()
  , updateState = \_ -> pure ()
  , rule = \p -> do
      message_contracts <- T.query @User.Message
      let messages = map snd message_contracts
      debug $ "Messages so far: " <> show (length messages)
      let lastMessage = List.maximumOn (.receivedAt) messages
      debug $ "Last message: " <> show lastMessage
      whenSome lastMessage $ \m ->
        when (m.receiver == p) $ do
          users <- T.query @User.User
          debug users
          let isSender = (\user -> user.username == m.sender)
          let replyTo = List.head $ filter (\(_, user) -> isSender user) users
          whenSome replyTo $ \(sender, _) ->
            T.dedupExercise sender (User.SendMessage p "Please, tell me more
↪about that.")
  , registeredTemplates = T.AllInDar
  , heartbeat = None
  }
```

Refresh `daml start` by pressing `r` (followed by `Enter` on Windows) in its terminal, then start the trigger with:

```
daml trigger --dar .daml/dist/gsg-trigger-0.1.0.dar \
             --trigger-name ChatBot:autoReply \
             --ledger-host localhost \
             --ledger-port 6865 \
             --ledger-user "bob"
```

Play a bit with `alice` and `bob` in your browser, to get a feel for how the trigger works. Watch both the messages in-browser and the debug statements printed by the trigger runner.

Let's walk through the `rule` code line-by-line:

> We use the `query` function to get all of the `Message` templates visible to the current party (`p`; in our case this will be `bob`). Per the documentation, this returns a list of tuples (contract id, payload), which we store as `message_contracts`.
> We then map the snd function on the result to get only the payloads, i.e. the actual data of the messages we can see.
> We print, as a `debug` message, the number of messages we can see.
> On the next line, get the message with the highest `receivedAt` field (maximumOn).

We then print another debug message, this time printing the message our code has identified as the last message visible to the current party . If you run this, you'll see that `lastMessage` is actually a `Optional Message`. This is because the maximumOn function will return the element from a list for which the given functions produces the highest value *if* the list has at least one element, but it needs to still do something sensible if the list is empty; in this case, it would return `None`.
When `lastMessage` is `Some m` (whenSome), we execute the given function. Otherwise, `lastMessage` is `None` and we implicitly do nothing.
Next, we need to check whether the message has been sent *to* or *by* the party running the trigger (with the current Daml model, it has to be one or the other, as messages are only visible to the sender and receiver). when the expression `m.receiver == p` is `True`, our expectations of the ledger state are wrong and we need to correct it. Otherwise, the state matches our rule and we don't need to do anything.
At this point we know the state is wrong , per our expectations, and start engaging in correcting actions. For this trigger, this means sending a message to the sender of the last message. In order to do that, we need to find the `User` contract for the sender. We start by getting the list of all `User` contracts we know about, which will be all users who follow the party running the trigger (and that party's own `User` contract). As for `Message` contracts earlier, the result of `query @User` is going to be a list of tuples with (contract id, payload). The big difference is that this time we actually want to keep the contract ids, as that is what we'll use to send a message back.
We print the list of users we just fetched, as a debug message.
We create a function `isSender` to identify the user we are looking for.
We get the user contract by applying our `isSender` function as a filter on the list of users, and then taking the head of that list, i.e. its first element.
Just like `maximumOn`, `head` will return an `Optional a`, so the next step is to check whether we have actually found the relevant `User` contract. In most cases we should find it, but remember that users can send us a message if *we* follow *them*, whereas we can only answer if *they* follow *us*.
If we did find some `User` contract to reply to, we extract the corresponding contract id (first element of the tuple, `sender`) and discard the payload (second element, `_`), and we exercise the `SendMessage` choice, passing in the current party `p` as the sender. See below for additional information on what that `dedup` in the name of the command means.

Command Deduplication

Daml Triggers react to many things, and it's usually important to make sure that the same command is not sent multiple times.

For example, in our `autoReply` chatbot above, the rule will be triggered not only when we receive a message, but also when we send one, as well as when we follow a user or get followed by a user, and when we stop following a user or a user stops following us.

It's easy to imagine a sequence of events that would make a naive trigger implementation send too many messages. For example:

`alice` sends `"hi"`, so the trigger runs and sends an `exercise` command.
Before the `exercise` command is fully processed, `carol` follows `bob`, which triggers the rule again. The state of all the `Message` contracts `bob` can see has not changed, so the rule might send the response to `alice` again.

We obviously don't want that to happen, as it would likely prevent us from passing that Turing test

we were after.

Triggers offer a few features to help users manage that. Possibly the simplest one is the `dedup*` family of ledger operations. When using those, the trigger runner will keep track of the commands currently sent and prevent sending the exact same command again. In the above example, the trigger would see that, when `carol` follows `bob` and the rule runs `dedupExercise`, there is already an Exercise command in flight with the exact same value, in this case same message, same sender and same receiver.

Note that, if instead the in-between event is `alice` following `carol`, this simple deduplication mechanism might not work as expected: because the `User` contract ID for `alice` would have changed, the new command is not the same as the in-flight one and thus a second `SendMessage` exercise would be sent to the ledger.

Similarly, if `alice` sends a second message quickly after the first one, this deduplication would prevent it, because the response does not have any reference to which message it's responding to. This may or may not be what we want.

If this simple deduplication is not suited to your use-case, you have two other tools at your disposal. The first one is the second argument to the `emitCommands` action (doc), which is a list of contract IDs. These IDs will be filtered out of any ACS `query` made by this trigger until the commands submitted as part of the same `emitCommands` call have completed. If your trigger is based on seeing certain contracts, this can be a simple, effective way to prevent triggering it multiple times.

The last tool you have at your disposal is the `getCommandsInflight` action (doc), which returns all of the commands this instance of the trigger runner has sent and that have not yet been resolved (i.e. either committed or failed). You can then build your own logic based on this list, the ACS, and possibly your own trigger state.

Finally, do keep in mind that all of these mechanisms rely on internal state from the trigger runner, which keeps track of which commands it has sent and for which it's not seen a completion. They will all fail to deduplicate if that internal state is lost, e.g. if the trigger runner is shut down and a new one is started. As such, these deduplication mechanisms should be seen as an optimization rather than a requirement for correctness. The Daml model should be designed such that duplicated commands are either rejected (e.g. using keys or relying on changing contract IDs) or benign.

Authorization

When using Daml triggers against a Ledger with *request authorization*, you can pass `--access-token-file token.jwt` to `daml trigger` which will read the token from the file `token.jwt`.

If you plan to run more than one trigger at a time, or triggers for more than one party at a time, you may be interested in the *Trigger Service*.

When Not to Use Daml Triggers

Daml triggers deliberately only allow you to express automation that listens for ledger events and reacts to them by sending commands to the ledger.

Daml Triggers are not suited for automation that needs to interact with services or data outside of the ledger. For those cases, you can write a ledger client using the JavaScript bindings running against the HTTP JSON API or the *Java bindings* running against the gRPC Ledger API.

Trigger Service

The *Run a No-Op Trigger* section shows a simple method using the `daml trigger` command to arrange for the execution of a single trigger. Using this method, a dedicated process is launched to host the trigger.

Complex workflows can require running many triggers for many parties and at a certain point, use of `daml trigger` with its process-per-trigger model becomes unwieldy. The Trigger Service provides the means to host multiple triggers for multiple parties running against a common ledger in a single process and provides a convenient interface for starting, stopping and monitoring them.

The Trigger Service is a ledger client that acts as an end-user agent. The Trigger Service intermediates between the ledger and end-users by running triggers on their behalf. The Trigger Service is an HTTP service. All requests and responses use JSON to encode data.

Start the Trigger Service

In this example, it is assumed there is a Ledger API server running on port 6865 on *localhost*.

```
daml trigger-service --config trigger-service.conf
```

The following snippet provides an example of what a possible *trigger-service.conf* configuration file could look like, alongside a few annotations with regards to the meaning of the configuration keys and possibly their default values.

```
{
  // Paths to the DAR files containing the code executed by the trigger.
  dar-paths = [
    "./my-app.dar"
  ]

  // Host address that the Trigger Service listens on. Defaults to 127.0.0.1.
  address = "127.0.0.1"

  // Trigger Service port number. Defaults to 8088.
  // A port number of 0 will let the system pick an ephemeral port.
  port = 8088
  // Optional. If using 0 as the port number, consider specifying the path to a
↪`port-file` where the chosen port will be saved in textual format.
  //port-file = "/path/to/port-file"

  // Mandatory. Ledger API server address and port.
  ledger-api {
```

(continues on next page)

(continued from previous page)

```
    address = "localhost"
    port = 6865
  }

  // Maximum inbound message size in bytes. Defaults to 4194304 (4 MB).
  max-inbound-message-size = 4194304

  // Minimum and maximum time interval before restarting a failed trigger.
↪Defaults to 5 and 60 seconds respectively.
  min-restart-interval = 5s
  max-restart-interval = 60s

  // Maximum HTTP entity upload size in bytes. Defaults to 4194304 (4 MB).
  max-http-entity-upload-size = 4194304

  // HTTP entity upload timeout. Defaults to 60 seconds.
  http-entity-upload-timeout = 60s

  // Use static or wall-clock time. Defaults to `wall-clock`.
  time-provider-type = "wall-clock"

  // Compiler configuration type to use between `default` or `dev`. Defaults to
↪`default`.
  compiler-config = "default"

  // Time-to-live used for commands emitted by the trigger. Defaults to 30
↪seconds.
  ttl = 30s

  // If true, initialize the database and terminate immediately. Defaults to
↪false.
  init-db = "false"

  // Do not abort if there are existing tables in the database schema. EXPERT
↪ONLY. Defaults to false.
  allow-existing-schema = "false"

  // Configuration of trigger runners.
  trigger-config {
    // The number of ledger client command invocations each trigger will attempt
↪to execute in parallel. Defaults to 8.
    parallelism = 8

    // Maximum number of retries for a failing ledger API command submission.
↪Failed submission requests may be
    // handled by trigger rules. Defaults to 6.
    max-retries = 6

    // Used to control maximum rate at which we perform ledger client submission
↪requests.
    max-submission-requests = 100 // Defaults to 100.
    max-submission-duration = 5s  // Defaults to 5s.

    // Size of the queue holding ledger API command submission failures. When
↪queue is filled, submission requests
    // are dropped. Defaults to 264.
```

(continues on next page)

(continued from previous page)

```
    submission-failure-queue-size = 264
  }

  // Configuration for the persistent store that will be used to keep track of
↪running triggers across restarts.
  // Mandatory if `init-db` is true. Otherwise optional. If not provided, the
↪trigger state will not be persisted
  // and restored across restarts.
  trigger-store {

    // Mandatory. Database coordinates.
    user = "postgres"
    password = "password"
    driver = "org.postgresql.Driver"
    url = "jdbc:postgresql://localhost:5432/test?&ssl=true"

    // Prefix for table names to avoid collisions. EXPERT ONLY. By default, this
↪is empty and not used.
    //table-prefix = "foo"

    // Maximum size for the database connection pool. Defaults to 8.
    pool-size = 8

    // Minimum idle connections for the database connection pool. Defaults to 8.
    min-idle = 8

    // Idle timeout for the database connection pool. Defaults to 10 seconds.
    idle-timeout = 10s

    // Timeout for database connection pool. Defaults to 5 seconds.
    connection-timeout = 5s
  }

  authorization {

    // Auth client to redirect to login. Defaults to `no`.
    auth-redirect = "no"

    // The following options configure the auth URIs.
    // Either just `auth-common-uri` or both `auth-internal-uri` and `auth-
↪external-uri` must be specified.
    // If all are specified, `auth-internal-uri` and `auth-external-uri` take
↪precedence.

    // Sets both the internal and external auth URIs.
    //auth-common-uri = "https://oauth2/common-uri"

    // Internal auth URI used by the Trigger Service to connect directly to the
↪Auth Middleware.
    auth-internal-uri = "https://oauth2/internal-uri"

    // External auth URI (the one returned to the browser).
    // This value takes precedence over the one specified for `auth-common`.
    auth-external-uri = "https://oauth2/external-uri"

    // Optional. URI to the auth login flow callback endpoint `/cb`. By default
↪it is constructed from the incoming login request.
```

(continues on next page)

(continued from previous page)

```
    // auth-callback-uri = "https://oauth2/callback-uri"

    // Maximum number of pending authorization requests. Defaults to 250.
    max-pending-authorizations = 250

    // Authorization timeout. Defaults to 60 seconds.
    authorization-timeout = 60s
  }

  // Optional. Trigger service ledger API client TLS configuration. By default
↪TLS configuration is disabled.
  //tls-config {
  //   enabled = "true"
  //
  //   // the certificate to be used by the server
  //   cert-chain-file = "/path/to/participant.crt"
  //
  //   // private key of the server
  //   private-key-file = "/path/to/participant.pem"
  //
  //   // trust collection, which means that all client certificates that will be
↪verified using the trusted
  //   // certificates in this store. If omitted, the JVM default trust store is
↪used.
  //   trust-collection-file = "/path/to/root-ca.crt"
  //}
}
```

The Trigger Service can also be started using command line arguments as shown below. The command `daml trigger-service --help` lists all available parameters.

Note: Using the configuration format shown above is the recommended way to configure Trigger Service, running with command line arguments is now deprecated.

```
daml trigger-service --ledger-host localhost \
                     --ledger-port 6865 \
                     --wall-clock-time
```

Although, as we'll see, the Trigger Service exposes an endpoint for end-users to upload DAR files to the service it is sometimes convenient to start the service pre-configured with a specific DAR. To do this, the `--dar` option is provided.

```
daml trigger-service --ledger-host localhost \
                     --ledger-port 6865 \
                     --wall-clock-time \
                     --dar .daml/dist/create-daml-app-0.1.0.dar
```

Endpoints

Start a Trigger

Start a trigger. In this example, `alice` starts the trigger called `trigger` in a module called `TestTrigger` of a package with ID `312094804c1468e2166bae3c9ba8b5cc0d285e31356304a2e9b0ac549df59d14`. The response contains an identifier for the running trigger that `alice` can use in subsequent commands involving the trigger.

HTTP Request

URL: `/v1/triggers`
Method: `POST`
Content-Type: `application/json`
Content:

```
{
  "triggerName":
↪"312094804c1468e2166bae3c9ba8b5cc0d285e31356304a2e9b0ac549df59d14:TestTrigger:trigger
↪",
  "party": "alice",
  "applicationId": "my-app-id"
}
```

where

`triggerName` contains the identifier for the trigger in the form `${packageId}:${moduleName}:${identifierName}`. You can find the package ID using `daml damlc inspect path/to/trigger.dar | head -1`.
`party` is the party on behalf of which the trigger is running.
`applicationId` is an optional field to specify the application ID the trigger will use for command submissions. If omitted, the trigger will default to using its random UUID identifier returned in the start request as the application ID.

HTTP Response

```
{
  "result":{"triggerId":"4d539e9c-b962-4762-be71-40a5c97a47a6"},
  "status":200
}
```

Stop a Trigger

Stop a running trigger. In this example, the request asks to stop the trigger started above.

HTTP Request

URL: `/v1/triggers/:id`
Method: `DELETE`
Content-Type: `application/json`
Content:

HTTP Response

Content-Type: `application/json`
Content:

```
{
  "result": {"triggerId":"4d539e9c-b962-4762-be71-40a5c97a47a6"},
  "status":200
}
```

List Running Triggers

List the triggers running on behalf of a given party.

HTTP Request

URL: `/v1/triggers?party=:party`
Method: `GET`

HTTP Response

Content-Type: `application/json`
Content:

```
{
  "result": {"triggerIds":["4d539e9c-b962-4762-be71-40a5c97a47a6"]},
  "status":200
}
```

Status of a Trigger

This endpoint returns data about a trigger, including the party on behalf of which it is running, its identifier, and its current state (querying the active contract set, running, or stopped).

HTTP Request

URL: `/v1/triggers/:id`
Method: `GET`

HTTP Response

Content-Type: `application/json`
Content:

```
{
  "result":
    {
      "party": "Alice",
      "triggerId":
↪"312094804c1468e2166bae3c9ba8b5cc0d285e31356304a2e9b0ac549df59d14:TestTrigger:trigger
↪",
      "status": "running"
    },
  "status":200
}
```

Upload a New DAR

Upload a DAR containing one or more triggers. If successful, the DAR's main package ID will be in the response (the main package ID for a DAR can also be obtained using `daml damlc inspect path/to/dar | head -1`).

HTTP Request

URL: `/v1/packages`
Method: `POST`
Content-Type: `multipart/form-data`
Content:
`dar=$dar_content`

HTTP Response

Content-Type: `application/json`
Content:

```
{
  "result": {"mainPackageId":
↪"312094804c1468e2166bae3c9ba8b5cc0d285e31356304a2e9b0ac549df59d14"},
  "status": 200
}
```

Liveness Check

This can be used as a liveness probe, e.g., in Kubernetes.

HTTP Request

URL: `/livez`
Method: `GET`

HTTP Response

A status code of `200` indicates a successful liveness check.

Content-Type: `application/json`
Content:

```
{ "status": "pass" }
```

Readiness Check

This can be used as a readiness probe, e.g., in Kubernetes.

HTTP Request

URL: `/readyz`
Method: `GET`

HTTP Response

A status code of `200` indicates a successful readiness check.

Metrics

Enable and Configure Reporting

To enable metrics and configure reporting, you can use the below config block in application config:

```
metrics {
  // Start a metrics reporter. Must be one of "console", "csv:///PATH",
↪"graphite://HOST[:PORT][/METRIC_PREFIX]", or "prometheus://HOST[:PORT]".
  reporter = "prometheus://localhost:9000"
  // Set metric reporting interval, examples: 1s, 30s, 1m, 1h
  reporting-interval = 30s
}
```

Reported Metrics

If a Prometheus metrics reporter is configured, the Trigger Service exposes the *common HTTP metrics* for all endpoints.

Authorization

The trigger service issues commands to the ledger that may require authorization through an access token. See *Ledger Authorization* for a description of authentication and authorization on Daml ledgers. How to obtain an access token is defined by the ledger operator. The trigger service interfaces with an *Auth Middleware* to obtain an access token in order to decouple it from the specific authentication and authorization mechanism used for a given ledger. The documentation includes an *Example Configuration using Auth0*.

Enable Authorization

You can use the following command-line flags to configure the trigger service to interface with a given auth middleware.

`--auth` The URI to the auth middleware. The auth middleware should be reachable under this URI from the client as well as the trigger service itself.

`--auth-callback` The login workflow may require redirection to the callback endpoint of the trigger service. This flag configures the URI to the trigger service's `/cb` endpoint, it should be reachable from the client.

For example, use the following flags if the trigger service and the auth middleware are both running behind a reverse proxy.:

```
--auth https://example.com/auth
--auth-callback https://example.com/trigger/cb
```

Assuming that the auth middleware is available under `https://example.com/auth` and the trigger service is available under `https://example.com/trigger`.

Note that the trigger service must be able to share cookies with the auth middleware as described in the *Deployment notes*.

Obtain Authorization

The trigger service will respond with 401 Unauthorized if a request requires authentication and authorization of the user. The trigger service can be configured to redirect to the `/login` endpoint via HTTP redirect (302 Found) using the command-line flag `-auth-redirect`. This can be useful for testing if the IAM does not require user input.

The 401 Unauthorized response will include a WWW-Authenticate header of the form:

```
WWW-Authenticate
    DamlAuthMiddleware realm=":claims",login=":login",auth=":auth"
```

where

`claims` are the required *Daml Ledger Claims*.
`login` is the URL to initiate the login flow on the auth middleware.
`auth` is the URL to check whether authorization has been granted.

The response will also include an entity with

Content-Type: `application/json`
Content:

```
{
    "realm": ":claims",
    "login": ":auth",
    "auth": ":login",
}
```

An application can direct the user to the login URL, wait until authorization has been granted, and repeat the original request once authorization has been granted. The auth URL can be used to poll until authorization has been granted. Alternatively, it can append a custom `redirect_url` parameter to the login URL and redirect to the resulting URL. Note that login with the IAM may require entering credentials into a web-form, i.e. the login URL should be opened in a web browser.

Example

This section describes how a web frontend can interact with the trigger service when authorization is required. Note, to avoid cross-origin requests and to enable sharing of cookies the web application and auth middleware should be exposed under the same domain, e.g. behind a shared reverse proxy.

Let's start with a request to the *list running triggers* endpoint.

```
const resp = await fetch("/trigger/v1/triggers?party=Alice");
if (resp.status >= 200 && resp.status < 300) {
    const result = await resp.json();
    // process result ...
} else if (resp.status === 401) {
    // handle Unauthorized ...
} else {
    // handle other error ...
}
```

If the request succeeds it decodes the JSON response body and continues processing the result, otherwise it checks if the request failed with 401 Unauthorized or another error. We will ignore the general error case and focus only on handling the Unauthorized response.

Login via Redirect

A simple solution is to redirect the browser to the login URL after adding a `redirect_url` parameter that points back to the current page.

```
const challenge = await resp.json();
var loginUrl = new URL(challenge.login);
loginUrl.searchParams.append("redirect_uri", window.location.href);
window.location.replace(loginUrl.href);
```

This code first decodes the JSON encoded authentication challenge included in the response body, then it extends the login URL with a `redirect_uri` parameter that points back to the current page, and redirects the browser to the login flow. The browser will be redirected to the original page after the login flow completed at which point authorization should have been granted and the original request should succeed.

Login via Popup

Another solution is to direct the user to the login page in a separate window, wait until authorization has been granted, and then retry the original request.

```
const challenge = await resp.json();
await popupLogin(challenge.login, challenge.auth);
// retry original request ...
```

The function `popupLogin` opens the login URL in a popup window and polls on the auth URL until authorization has been granted. It raises an error if the login window closes before authorization has been granted.

```
function popupLogin(login, auth) {
    return new Promise(function (resolve, reject) {
        var popup = window.open(login);
        var timer = setInterval(async function() {
            const closed = popup.closed;
            const resp = await fetch(auth);
            if (resp.status >= 200 && resp.status < 300) {
                // The user logged in
                clearInterval(timer);
                popup.close();
                resolve();
            } else if (closed) {
                // The popup is closed but we are not logged in.
                reject(new Error("Login failed"))
            }
        }, 1000);
    });
}
```

Auth Middleware

Daml ledgers only validate authorization tokens. The issuance of those tokens however is something defined by the participant operator and can vary significantly across deployments. This poses a challenge when developing applications that need to be able to acquire and refresh authorization tokens but don't want to tie themselves to any particular mechanism for token issuance. The Auth Middleware aims to address this problem by providing an API that decouples Daml applications from these details. The participant operator can provide an Auth Middleware that is suitable for their authentication and authorization mechanism. Daml includes an implementation of an Auth Middleware that supports OAuth 2.0 Authorization Code Grant. If this implementation is not compatible with your mechanism for token issuance, you can implement your own Auth Middleware provided it conforms to the same API.

Features

The Auth Middleware is designed to fulfill the following goals:

Be agnostic of the authentication and authorization protocol required by the identity and access management (IAM) system used by the participant operator.
Allow fine grained access control via Daml ledger claims.
Support token refresh for long running clients that should not require user interaction.

Auth Middleware API

An implementation of the Auth Middleware must provide the following API.

Obtain Access Token

The application contacts this endpoint to determine if the issuer of the request is authenticated and authorized to access the given claims. The application must forward any cookies that it itself received in the original request. The response will contain an access token and optionally a refresh token if the issuer of the request is authenticated and authorized. Otherwise, the response will be 401 Unauthorized.

HTTP Request

URL: `/auth?claims=:claims`
Method: `GET`
Headers: `Cookie`

where

`claims` are the requested *Daml Ledger Claims*.

For example:

```
/auth?claims=actAs:Alice+applicationId:MyApp
```

Note: When using user management, the participant operator may have configured their IAM to issue user tokens. The Auth Middleware currently doesn't accept an input parameter specific to user IDs. As such, it is up to the IAM to map claims request to the required user token. Our recommendation to participant operators is to map the `applicationId` claim to the required user ID. Application developers should contact their ledger operator to understand how they are supposed to request for a token.

HTTP Response

```
{
    "access_token": "...",
    "refresh_token": "..."
}
```

where

`access_token` is the access token to use for Daml ledger commands.
`refresh_token` (optional) can be used to refresh an expired access token on the `/refresh` endpoint.

Request Authorization

The application directs the user to this endpoint if the `/auth` endpoint returned 401 Unauthorized. This will request authentication and authorization of the user from the IAM for the given claims. E.g. in the OAuth 2.0 based implementation included in Daml, this will start an Authorization Code Grant flow.

If authorization is granted this will store the access and optional refresh token in a cookie. The request can define a callback URI, if specified this endpoint will redirect to the callback URI at the end of the flow. Otherwise, it will respond with a status code that indicates whether authorization was successful or not.

HTTP Request

URL: `/login?claims=:claims&redirect_uri=:redirect_uri&state=:state`
Method: `GET`

where

`claims` are the requested *Daml Ledger Claims*.
`redirect_uri` (optional) redirect to this URI at the end of the flow. Passes `error` and optionally `error_description` parameters if authorization failed.
`state` (optional) forward this parameter to the `redirect_uri` if specified.

For example:

```
/login?claims=actAs:Alice+applicationId:MyApp&redirect_uri=http://example.com/cb&
↪state=2b56cc2e-01ad-4e51-a9b3-124d4bbe0a91
```

Refresh Access Token

The application contacts this endpoint to refresh an expired access token without requiring user input. Token refresh is available if the `/auth` endpoint return a refresh token along side the access token. This endpoint will return a new access token and optionally a new refresh token to replace the old.

HTTP Request

URL: `/refresh`
Method: `POST`
Content-Type: `application/json`
Content:

```
{
  "refresh_token": "..."
}
```

where

`refresh_token` is the refresh token returned by `/auth` or a previous `/refresh` request.

HTTP Response

```
{
    "access_token": "...",
    "refresh_token": "..."
}
```

where

`access_token` is the access token to use for Daml ledger commands.
`refresh_token` (optional) can be used to refresh an expired access token on the `/refresh` endpoint.

Daml Ledger Claims

A list of claims specifies the set of capabilities that are requested. These are passed as a URL-encoded, space-separated list of individual claims of the following form:

admin Access to admin-level services.
readAs:<Party Name> Read access for the given party.
actAs:<Party Name> Issue commands on behalf of the given party.
applicationId:<Application Id> Restrict access to commands issued with the given application ID.

See *Access Tokens and Claims* for further information on Daml ledger capabilities.

OAuth 2.0 Auth Middleware

Daml includes an implementation of an auth middleware that supports OAuth 2.0 Authorization Code Grant. The implementation aims to be configurable to support different OAuth 2.0 providers and to allow custom mappings from Daml ledger claims to OAuth 2.0 scopes.

OAuth 2.0 Configuration

RFC 6749 specifies that OAuth 2.0 providers offer two endpoints: The authorization endpoint and the token endpoint. The URIs for these endpoints can be configured independently using the following fields:

```
oauth-auth
oauth-token
```

The OAuth 2.0 provider may require that the application identify itself using a client identifier and client secret. These can be specified using the following environment variables:

```
DAML_CLIENT_ID
DAML_CLIENT_SECRET
```

The auth middleware assumes that the OAuth 2.0 provider issues JWT access tokens. The `/auth` endpoint will validate the token, if available, and ensure that it grants the requested claims. The auth middleware accepts the same command-line flags as the *Daml Sandbox* to define the public key for token validation.

Request Templates

The exact format of OAuth 2.0 requests may vary between providers. Furthermore, the mapping from Daml ledger claims to OAuth 2.0 scopes is defined by the IAM operator. For that reason OAuth 2.0 requests made by auth middleware can be configured using user defined Jsonnet templates. Templates are parameterized configurations expressed as top-level functions.

Authorization Request

This template defines the format of the Authorization request. Use the following config field to use a custom template:

`oauth-auth-template`

Arguments

The template will be passed the following arguments:

`config` (object)
- `clientId` (string) the OAuth 2.0 client identifier
- `clientSecret` (string) the OAuth 2.0 client secret

`request` (object)
- **`claims` (object) the requested claims**
  * `admin` (bool)
  * `applicationId` (string or null)
  * `actAs` (list of string)
  * `readAs` (list of string)
- `redirectUri` (string)
- `state` (string)

Returns

The query parameters for the authorization endpoint encoded as an object with string values.

Example

```
local scope(claims) =
  local admin = if claims.admin then "admin";
  local applicationId = if claims.applicationId != null then "applicationId:" +
↪claims.applicationId;
  local actAs = std.map(function(p) "actAs:" + p, claims.actAs);
  local readAs = std.map(function(p) "readAs:" + p, claims.readAs);
  [admin, applicationId] + actAs + readAs;

function(config, request) {
  "audience": "https://daml.com/ledger-api",
  "client_id": config.clientId,
  "redirect_uri": request.redirectUri,
```

(continues on next page)

(continued from previous page)

```
  "response_type": "code",
  "scope": std.join(" ", ["offline_access"] + scope(request.claims)),
  "state": request.state,
}
```

Token Request

This template defines the format of the Token request. Use the following config field to use a custom template:

`oauth-token-template`

Arguments

The template will be passed the following arguments:

`config` (object)
- `clientId` (string) the OAuth 2.0 client identifier
- `clientSecret` (string) the OAuth 2.0 client secret

`request` (object)
- `code` (string)
- `redirectUri` (string)

Returns

The request parameters for the token endpoint encoded as an object with string values.

Example

```
function(config, request) {
  "client_id": config.clientId,
  "client_secret": config.clientSecret,
  "code": request.code,
  "grant_type": "authorization_code",
  "redirect_uri": request.redirectUri,
}
```

Refresh Request

This template defines the format of the Refresh request. Use the following config field to use a custom template:

`oauth-refresh-template`

Arguments

The template will be passed the following arguments:

`config` (object)
- `clientId` (string) the OAuth 2.0 client identifier
- `clientSecret` (string) the OAuth 2.0 client secret

`request` (object)
- `refreshToken` (string)

Returns

The request parameters for the authorization endpoint encoded as an object with string values.

Example

```
function(config, request) {
  "client_id": config.clientId,
  "client_secret": config.clientSecret,
  "grant_type": "refresh_code",
  "refresh_token": request.refreshToken,
}
```

Deployment Notes

The auth middleware API relies on sharing cookies between the auth middleware and the Daml application. One way to enable this is to expose the auth middleware and the Daml application under the same domain, e.g. through a reverse proxy. Note that you will need to specify the external callback URI in that case using the `--callback` command-line flag.

For example, assuming the following nginx configuration snippet:

```
http {
  server {
    server_name example.com
    location /auth/ {
      proxy_pass http://localhost:3000/;
    }
  }
}
```

You would invoke the OAuth 2.0 auth middleware with the following flags:

```
oauth2-middleware \
    --config oauth-middleware.conf
```

The required config would look like

```
{
  // Environment variables:
```

(continues on next page)

(continued from previous page)

```
  // DAML_CLIENT_ID      The OAuth2 client-id - must not be empty
  // DAML_CLIENT_SECRET  The OAuth2 client-secret - must not be empty
  client-id = ${DAML_CLIENT_ID}
  client-secret = ${DAML_CLIENT_SECRET}

  //IP address that OAuth2 Middleware service listens on. Defaults to 127.0.0.1.
  address = "127.0.0.1"
  //OAuth2 Middleware service port number. Defaults to 3000. A port number of 0
↪will let the system pick an ephemeral port. Consider specifying `--port-file`
↪option with port number 0.
  port = 3000

  //URI to the auth middleware's callback endpoint `/cb`. By default constructed
↪from the incoming login request.
  callback-uri = "https://example.com/auth/cb"

  //Maximum number of simultaneously pending login requests. Requests will be
↪denied when exceeded until earlier requests have been completed or timed out.
  max-login-requests = 250

  //Login request timeout. Requests will be evicted if the callback endpoint
↪receives no corresponding request in time.
  login-timeout = 60s

  //Enable the Secure attribute on the cookie that stores the token. Defaults to
↪true. Only disable this for testing and development purposes.
  cookie-secure = "true"

  //URI of the OAuth2 authorization endpoint
  oauth-auth="https://oauth2-provider.com/auth_uri"

  //URI of the OAuth2 token endpoint
  oauth-token="https://oauth2-provider.com/token_uri"

  //OAuth2 authorization request Jsonnet template
  oauth-auth-template="file://path/oauth/auth/template"

  //OAuth2 token request Jsonnet template
  oauth-token-template = "file://path/oauth/token/template"

  //OAuth2 refresh request Jsonnet template
  oauth-refresh-template = "file://path/oauth/refresh/template"

  // Enables JWT-based authorization, where the JWT is signed by one of the below
↪Jwt based token verifiers
  token-verifier {
    // type can be rs256-crt, es256-crt, es512-crt or rs256-jwks
    type = "rs256-jwks"
    // X509 certificate file (.crt)/JWKS url from where the public key is loaded
    uri = "https://example.com/.well-known/jwks.json"
  }
}
```

The oauth2-middleware can also be started using cli-args.

Note: Configuration file is the recommended way to run oauth2-middleware, running via cli-args is

now deprecated

```
oauth2-middleware \
    --callback https://example.com/auth/cb \
    --address localhost \
    --http-port 3000 \
    --oauth-auth https://oauth2-provider.com/auth_uri \
    --oauth-token https://oauth2-provider.com/token_uri \
    --auth-jwt-rs256-jwks https://example.com/.well-known/jwks.json
```

Some browsers reject `Secure` cookies on unencrypted connections even on localhost. You can pass the command-line flag `--cookie-secure no` for testing and development on localhost to avoid this.

Metrics

You may configure the oauth2-middleware to expose the *common HTTP metrics* via a Prometheus reporter by adding the below section to the application config:

```
metrics {
  // Start a metrics reporter. Must be one of "console", "csv:///PATH",
↪"graphite://HOST[:PORT][/METRIC_PREFIX]", or "prometheus://HOST[:PORT]".
  reporter = "prometheus://localhost:9000"
  // Set metric reporting interval , examples : 1s, 30s, 1m, 1h
  reporting-interval = 30s
}
```

Liveness and Readiness Endpoints

The following sections describe the endpoints that can be used to probe the liveness and readiness of the auth middleware service.

Liveness Check

This can be used as a liveness probe, e.g., in Kubernetes.

HTTP Request

URL: `/livez`
Method: `GET`

HTTP Response

A status code of `200` indicates a successful liveness check.

Content-Type: `application/json`
Content:

```
{ "status": "pass" }
```

Readiness Check

This can be used as a readiness probe, e.g., in Kubernetes.

HTTP Request

URL: `/readyz`
Method: `GET`

HTTP Response

A status code of `200` indicates a successful readiness check.

1.12.7 Errors

1.12.7.1 Command Deduplication

The interaction of a Daml application with the ledger is inherently asynchronous: applications send commands to the ledger, and some time later they see the effect of that command on the ledger. Many things can fail during this time window:

The application can crash.
The participant node can crash.
Messages can be lost on the network.
The ledger may be slow to respond due to a high load.

If you want to make sure that an intended ledger change is not executed twice, your application needs to robustly handle all failure scenarios. This guide covers the following topics:

How command deduplication works.
How applications can effectively use the command deduplication.

How Command Deduplication Works

The following fields in a command submissions are relevant for command deduplication. The first three form the *change ID* that identifies the intended ledger change.

> The union of *party* and *act_as* define the submitting parties.
> The *application ID* identifies the application that submits the command.
> The *command ID* is chosen by the application to identify the intended ledger change.
> The deduplication period specifies the period for which no earlier submissions with the same change ID should have been accepted, as witnessed by a completion event on the *command completion service*. If such a change has been accepted in that period, the current submission shall be rejected. The period is specified either as a *deduplication duration* or as a *deduplication offset* (inclusive).
> The *submission ID* is chosen by the application to identify a specific submission. It is included in the corresponding completion event so that the application can correlate specific submissions to specific completions. An application should never reuse a submission ID.

The ledger may arbitrarily extend the deduplication period specified in the submission, even beyond the maximum deduplication duration specified in the *ledger configuration*.

Note: The maximum deduplication duration is the length of the deduplication period guaranteed to be supported by the participant.

The deduplication period chosen by the ledger is the *effective deduplication period.* The ledger may also convert a requested deduplication duration into an effective deduplication offset or vice versa. The effective deduplication period is reported in the command completion event in the *deduplication duration* or *deduplication offset* fields.

A command submission is considered a **duplicate submission** if at least one of the following holds:

> The submitting participant's completion service contains a successful completion event for the same *change ID* within the *effective* deduplication period.
> The participant or Daml ledger are aware of another command submission in-flight with the same *change ID* when they perform command deduplication.

The outcome of command deduplication is communicated as follows:

> Command submissions via the *command service* indicate the command deduplication outcome as a synchronous gRPC response unless the gRPC deadline was exceeded.
>
> ---
>
> **Note:** The outcome MAY additionally appear as a completion event on the *command completion service*, but applications using the *command service* typically need not process completion events.
>
> ---
>
> Command submissions via the *command submission service* can indicate the outcome as a synchronous gRPC response, or asynchronously through the *command completion service*. In particular, the submission may be a duplicate even if the command submission service acknowledges the submission with the gRPC status code `OK`.

Independently of how the outcome is communicated, command deduplication generates the following outcomes of a command submission:

> If there is no conflicting submission with the same *change ID* on the Daml ledger or in-flight, the completion event and possibly the response convey the result of the submission (success

or a gRPC error; *Error Codes* explains how errors are communicated).
The gRPC status code `ALREADY_EXISTS` with error code ID *DUPLICATE_COMMAND* indicates that there is an earlier command completion for the same *change ID* within the effective deduplication period.
The gRPC status code `ABORTED` with error code id *SUBMISSION_ALREADY_IN_FLIGHT* indicates that another submission for the same *change ID* was in flight when this submission was processed.
The gRPC status code `FAILED_PRECONDITION` with error code id *INVALID_DEDUPLICATION_PERIOD* indicates that the specified deduplication period is not supported. The fields `longest_duration` or `earliest_offset` in the metadata specify the longest duration or earliest offset that is currently supported on the Ledger API endpoint. At least one of the two fields is present.
Neither deduplication durations up to the *maximum deduplication duration* nor deduplication offsets published within that duration SHOULD result in this error. Participants may accept longer periods at their discretion.
The gRPC status code `FAILED_PRECONDITION` with error code id *PARTICIPANT_PRUNED_DATA_ACCESSED*, when specifying a deduplication period represented by an offset, indicates that the specified deduplication offset has been pruned. The field `earliest_offset` in the metadata specifies the last pruned offset.

For deduplication to work as intended, all submissions for the same ledger change must be submitted via the same participant. Whether a submission is considered a duplicate is determined by completion events, and by default a participant outputs only the completion events for submissions that were requested via the very same participant.

How to Use Command Deduplication

To effectuate a ledger change exactly once, the application must resubmit a command if an earlier submission was lost. However, the application typically cannot distinguish a lost submission from slow submission processing by the ledger. Command deduplication allows the application to resubmit the command until it is executed and reject all duplicate submissions thereafter.

Some ledger changes can be executed at most once, so no command deduplication is needed for them. For example, if the submitted command exercises a consuming choice on a given contract ID, this command can be accepted at most once because every contract can be archived at most once. All duplicate submissions of such a change will be rejected with *CONTRACT_NOT_ACTIVE*.

In contrast, a *Create command* would create a fresh contract instance of the given *template* for each submission that reaches the ledger (unless other constraints such as the *template preconditions* or contract key uniqueness are violated). Similarly, an *Exercise command* on a non-consuming choice or an *Exercise-By-Key command* may be executed multiple times if submitted multiple times. With command deduplication, applications can ensure such intended ledger changes are executed only once within the deduplication period, even if the application resubmits, say because it considers the earlier submissions to be lost or forgot during a crash that it had already submitted the command.

Known Processing Time Bounds

For this strategy, you must estimate a bound `B` on the processing time and forward clock drifts in the Daml ledger with respect to the application's clock. If processing measured across all retries takes longer than your estimate `B`, the ledger change may take effect several times. Under this caveat, the following strategy works for applications that use the *Command Service* or the *Command Submission* and *Command Completion Service*.

Note: The bound `B` should be at most the configured *maximum deduplication duration*. Otherwise you rely on the ledger accepting longer deduplication durations. Such reliance makes your application harder to port to other Daml ledgers and fragile, as the ledger may stop accepting such extended durations at its own discretion.

---

1. Choose a command ID for the ledger change, in a way that makes sure the same ledger change is always assigned the same command ID. Either determine the command ID deterministically (e.g., if your contract payload contains a globally unique identifier, you can use that as your command ID), or choose the command ID randomly and persist it with the ledger change so that the application can use the same command ID in resubmissions after a crash and restart.

   Note: Make sure that you assign the same command ID to all command (re-)submissions of the same ledger change. This is useful for the recovery procedure after an application crash/restart. After a crash, the application in general cannot know whether it has submitted a set of commands before the crash. If in doubt, resubmit the commands using the same command ID. If the commands had been submitted before the crash, command deduplication on the ledger will reject the resubmissions.

   ---

2. When you use the *Command Completion Service*, obtain a recent offset on the completion stream `OFF1`, say the *current ledger end*.
3. Submit the command with the following parameters:
   Set the *command ID* to the chosen command ID from *Step 1*.
   Set the *deduplication duration* to the bound `B`.

   Note: It is prudent to explicitly set the deduplication duration to the desired bound `B`, to guard against the case where a ledger configuration update shortens the maximum deduplication duration. With the bound `B`, you will be notified of such a problem via an *INVALID_DEDUPLICATION_PERIOD* error if the ledger does not support deduplication durations of length `B` any more.
   If you omitted the deduplication period, the currently valid maximum deduplication duration would be used. In this case, a ledger configuration update could silently shorten the deduplication period and thus invalidate your deduplication analysis.

   Set the *submission ID* to a fresh value, e.g., a random UUID.
   Set the timeout (gRPC deadline) to the expected submission processing time (Command Service) or submission hand-off time (Command Submission Service).
   The **submission processing time** is the time between when the application sends off a submission to the *Command Service* and when it receives (synchronously, unless it times out) the acceptance or rejection. The **submission hand-off time** is the time between when the application sends off a submission to the *Command Submission Service* and when it obtains a synchronous response for this gRPC call. After the RPC timeout, the application considers the submission as lost and enters a retry loop. This timeout is typically much

shorter than the deduplication duration.

4. Wait until the RPC call returns a response.

   Status codes other than `OK` should be handled according to *error handling*.

   When you use the *Command Service* and the response carries the status code `OK`, the ledger change took place. You can report success.

   When you use the *Command Submission Service*, subscribe with the *Command Completion Service* for completions for `actAs` from `OFF1` (exclusive) until you see a completion event for the change ID and the submission ID chosen in *Step 3*. If the completion's status is `OK`, the ledger change took place and you can report success. Other status codes should be handled according to *error handling*.

   This step needs no timeout as the *Command Submission Service* acknowledges a submission only if there will eventually be a completion event, unless relevant parts of the system become permanently unavailable.

Error Handling

Error handling is needed when the status code of the command submission RPC call or in the *completion event* is not `OK`. The following table lists appropriate reactions by status code (written as `STATUS_CODE`) and error code (written in capital letters with a link to the error code documentation). Fields in the error metadata are written as `field` in lowercase letters.

Table 1: Command deduplication error handling with known processing time bound

Error condition	Reaction
`DEADLINE_EXCEEDED`	Consider the submission lost. Retry from *Step 2*, obtaining the completion offset `OFF1`, and possibly increase the timeout.
Application crashed	Retry from *Step 2*, obtaining the completion offset `OFF1`.
`ALREADY_EXISTS` / *DUPLICATE_COMMAND*	The change ID has already been accepted by the ledger within the reported deduplication period. The optional field `completion_offset` contains the precise offset. The optional field `existing_submission_id` contains the submission ID of the successful submission. Report success for the ledger change.
`FAILED_PRECONDITION` / *INVALID_DEDUPLICATION_PERIOD*	The specified deduplication period is longer than what the Daml ledger supports or the ledger cannot handle the specified deduplication offset. `earliest_offset` contains the earliest deduplication offset or `longest_duration` contains the longest deduplication duration that can be used (at least one of the two must be provided). Options: Negotiate support for longer deduplication periods with the ledger operator. Set the deduplication offset to `earliest_offset` or the deduplication duration to `longest_duration` and retry from *Step 2*, obtaining the completion offset `OFF1`. This may lead to accepting the change twice within the originally intended deduplication period.
`FAILED_PRECONDITION` / *PARTICIPANT_PRUNED_DATA_ACCESSED*	The specified deduplication offset has been pruned by the participant. `earliest_offset` contains the last pruned offset. Use the *Command Completion Service* by asking for the *completions*, starting from the last pruned offset by setting *offset* to the value of `earliest_offset`, and use the first received *offset* as a deduplication offset.
`ABORTED` / *SUBMISSION_ALREADY_IN_FLIGHT* This error occurs only as an RPC response, not inside a completion event.	There is already another submission in flight, with the submission ID in `existing_submission_id`. When you use the *Command Service*, wait a bit and retry from *Step 3*, submitting the command. Since the in-flight submission might still be rejected, (repeated) resubmission ensures that you (eventually) learn the outcome: If an earlier submission was accepted, you will eventually receive a *DUPLICATE_COMMAND* rejection. Otherwise, you have a second chance to get the ledger change accepted on the ledger and learn the outcome. When you use the *Command Completion Service*, look for a completion for `existing_submission_id` instead of the chosen submission ID in *Step 4*.
`ABORTED` / other error codes	Wait a bit and retry from *Step 2*, obtaining the completion offset `OFF1`.
other error conditions	Use background knowledge about the business workflow and the current ledger state to decide whether earlier submissions might still get accepted. If you conclude that it cannot be accepted any more, stop retrying and report that the ledger change failed. Otherwise, retry from *Step 2*, obtaining a completion offset `OFF1`, or give up without knowing for sure that the ledger change will not happen. For example, if the ledger change only creates a contract instance of a template, you can never be sure, as any outstanding submission might still be accepted.

Failure Scenarios

The above strategy can fail in the following scenarios:

1. The bound `B` is too low: The command can be executed multiple times.
   Possible causes:
   You have retried for longer than the deduplication duration, but never got a meaningful answer, e.g., because the timeout (gRPC deadline) is too short. For example, this can happen due to long-running Daml interpretation when using the *Command Service*.
   The application clock drifts significantly from the participant's or ledger's clock.
   There are unexpected network delays.
   Submissions are retried internally in the participant or Daml ledger and those retries do not stop before `B` is over. Refer to the specific ledger's documentation for more information.
2. Unacceptable changes cause infinite retries
   You need business workflow knowledge to decide that retrying does not make sense any more. Of course, you can always stop retrying and accept that you do not know the outcome for sure.

Unknown Processing Time Bounds

Finding a good bound `B` on the processing time is hard, and there may still be unforeseen circumstances that delay processing beyond the chosen bound `B`. You can avoid these problems by using deduplication offsets instead of durations. An offset defines a point in the history of the ledger and is thus not affected by clock skews and network delays. Offsets are arguably less intuitive and require more effort by the application developer. We recommend the following strategy for using deduplication offsets:

1. Choose a fresh command ID for the ledger change and the `actAs` parties, which (together with the application ID) determine the change ID. Remember the command ID across application crashes. (Analogous to *Step 1 above*)
2. Obtain a recent offset `OFF0` on the completion event stream and remember across crashes that you use `OFF0` with the chosen command ID. There are several ways to do so:
   Use the *Command Completion Service* by asking for the *current ledger end*.

   Note: Some ledger implementations reject deduplication offsets that do not identify a command completion visible to the submitting parties with the error code id *INVALID_DEDUPLICATION_PERIOD*. In general, the ledger end need not identify a command completion that is visible to the submitting parties. When running on such a ledger, use the Command Service approach described next.

   Use the *Command Service* to obtain a recent offset by repeatedly submitting a dummy command, e.g., a *Create-And-Exercise command* of some single-signatory template with the *Archive* choice, until you get a successful response. The response contains the *completion offset*.
3. When you use the *Command Completion Service*:
   If you execute this step the first time, set `OFF1 = OFF0`.
   If you execute this step as part of *error handling* retrying from Step 3, obtaining the completion offset `OFF1`, obtain a recent offset on the completion stream `OFF1`, say its current end. (Analogous to *step 2 above*)
4. Submit the command with the following parameters (analogous to *Step 3 above* except for the deduplication period):

Set the *command ID* to the chosen command ID from *Step 1*.
Set the *deduplication offset* to `OFF0`.
Set the *submission ID* to a fresh value, e.g., a random UUID.
Set the timeout (gRPC deadline) to the expected submission processing time (Command Service) or submission hand-off time (Command Submission Service).

5. Wait until the RPC call returns a response.
   Status codes other than `OK` should be handled according to *error handling*.
   When you use the *Command Service* and the response carries the status code `OK`, the ledger change took place. You can report success. The response contains a *completion offset* that you can use in *Step 2* of later submissions.
   When you use the *Command Submission Service*, subscribe with the *Command Completion Service* for completions for `actAs` from `OFF1` (exclusive) until you see a completion event for the change ID and the submission ID chosen in *step 3*. If the completion's status is `OK`, the ledger change took place and you can report success. Other status codes should be handled according to *error handling*.

Error Handling

The same as *for known bounds*, except that the former retry from *Step 2* becomes retry from *Step 3*.

Failure Scenarios

The above strategy can fail in the following scenarios:

1. No success within the supported deduplication period
   When the application receives a *INVALID_DEDUPLICATION_PERIOD* error, it cannot achieve exactly once execution any more within the originally intended deduplication period.
2. Unacceptable changes cause infinite retries
   You need business workflow knowledge to decide that retrying does not make sense any more. Of course, you can always stop retrying and accept that you do not know the outcome for sure.

1.12.8 Authorization

When developing Daml applications using SDK tools, your local setup will most likely not perform any Ledger API request authorization – by default, any valid Ledger API request will be accepted by the sandbox.

This is not the case for participant nodes of deployed ledgers. For every Ledger API request, the participant node checks whether the request contains an access token that is valid and sufficient to authorize that request. You thus need to add support for authorization using access tokens to your application to run it against a deployed ledger.

Note: In case of mutual (two-way) TLS authentication, the Ledger API client must present its certificate (in addition to an access token) to the Ledger API server as part of the authentication process. The provided certificate must be signed by a certificate authority (CA) trusted by the Ledger API server. Note that the identity of the application will not be proven by using this method, i.e. the *application_id* field in the request is not necessarily correlated with the CN (Common Name) in the certificate.

1.12.8.1 Introduction

Your Daml application sends requests to the *Ledger API* exposed by a participant node to submit changes to the ledger (e.g., *exercise choice X on contract Y as party Alice*), or to read data from the ledger (e.g., *read all active contracts visible to party Alice*). Your application might send these requests via a middleware like the *JSON API*.

Whether a participant node *can* serve such a request depends on whether the participant node hosts the respective parties, and whether the request is valid according to the *Daml Ledger Model*. Whether a participant node *will* serve such a request to a Daml application depends on whether the request includes an access token that is valid and sufficient to authorize the request for this participant node.

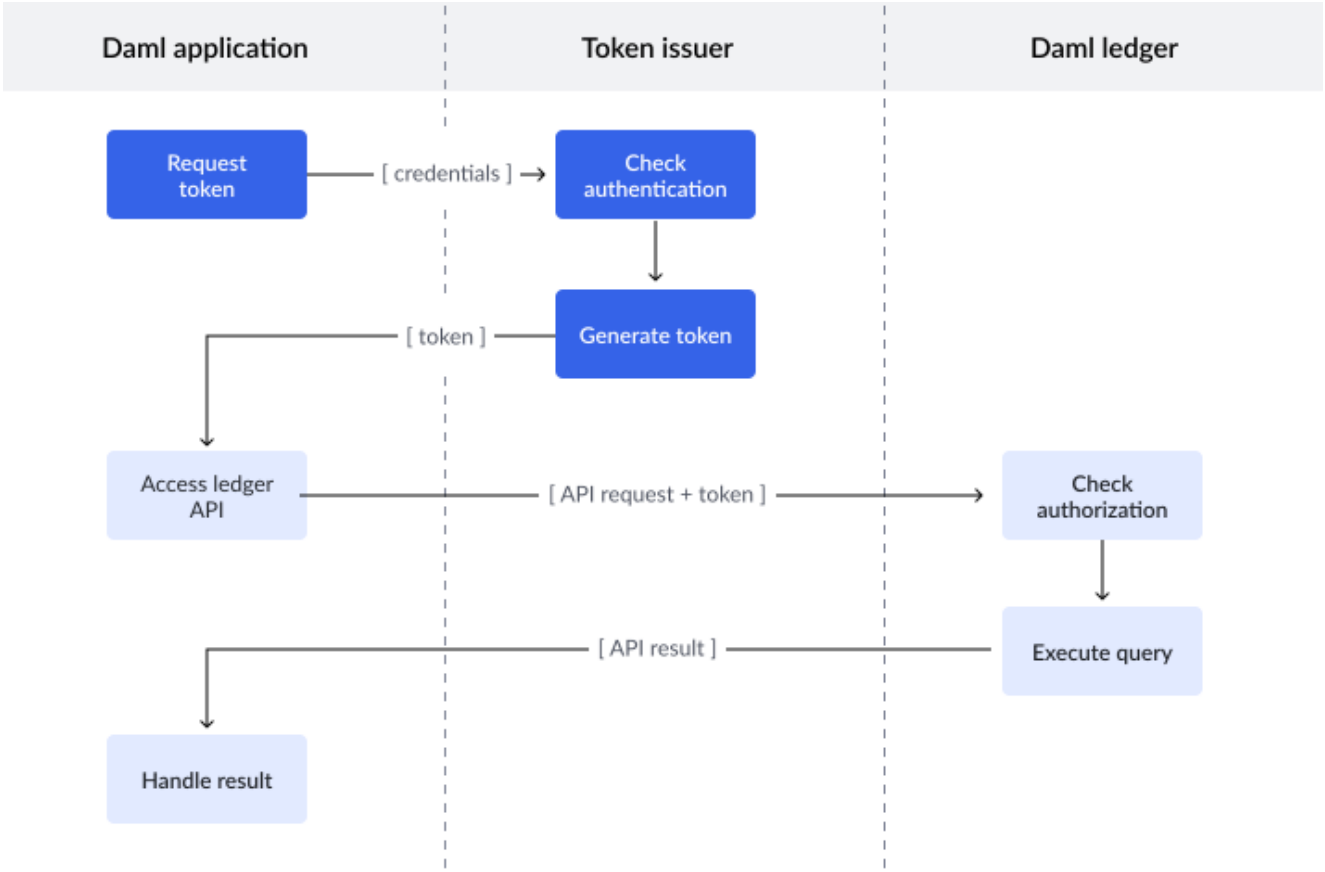
1.12.8.2 Acquire and Use Access Tokens

How an application acquires access tokens depends on the participant node it talks to and is ultimately set up by the participant node operator. Many setups use a flow in the style of OAuth 2.0.

In this scenario, the Daml application first contacts a token issuer to get an access token. The token issuer verifies the identity of the requesting application, looks up the privileges of the application, and generates a signed access token describing those privileges.

Once the access token is issued, the Daml application sends it along with every Ledger API request. The Daml ledger verifies:

that the token was issued by one of its trusted token issuers
that the token has not been tampered with
that the token had not expired
that the privileges described in the token authorize the request



How you attach tokens to requests depends on the tool or library you use to interact with the Ledger API. See the tool's or library's documentation for more information. (E.g. relevant documentation for the *Java bindings* and the *JSON API*.)

1.12.8.3 Access Tokens and Rights

Access tokens contain information about the rights granted to the bearer of the token. These rights are specific to the API being accessed.

The Daml Ledger API uses the following rights to govern request authorization:

`public`: the right to retrieve publicly available information, such as the ledger identity
`participant_admin`: the right to administer the participant node
`idp_admin`: the right to administer the users and parties belonging the same identity provider configuration as the authenticated user
`canReadAs(p)`: the right to read information off the ledger (like the active contracts) visible to the party `p`
`canActsAs(p)`: same as `canReadAs(p)`, with the added right of issuing commands on behalf of the party `p`

The following table summarizes the rights required to access each Ledger API endpoint:

Ledger API service	Endpoint	Required right
LedgerIdentityService	GetLedgerIdentity	public
ActiveContractsService	GetActiveContracts	for each requested party p: canReadAs(p)
CommandCompletionService	CompletionEnd	public
	CompletionStream	for each requested party p: canReadAs(p)
CommandSubmissionService	Submit	for submitting party p: canActAs(p)
CommandService	All	for submitting party p: canActAs(p)
EventQueryService	All	for each requesting party p: canReadAs(p)
Health	All	no access token required for health checking
IdentityProviderConfigService	All	participant_admin
LedgerConfigurationService	GetLedgerConfiguration	public
MeteringReportService	All	participant_admin
PackageService	All	public
PackageManagementService	All	participant_admin
PartyManagementService	All	participant_admin
	All (except GetParticipantId, UpdatePartyIdentityProviderId)	idp_admin
ParticipantPruningService	All	participant_admin
ServerReflection	All	no access token required for gRPC service reflection
TimeService	GetTime	public
	SetTime	participant_admin
TransactionService	LedgerEnd	public
	All (except LedgerEnd)	for each requested party p: canReadAs(p)
UserManagementService	All	participant_admin
	All (except UpdateUserIdentityProviderId)	idp_admin
	GetUser	authenticated users can get their own user
	ListUserRights	authenticated users can list their own rights
VersionService	All	public

1.12.8.4 Access Token Formats

Applications should treat access tokens as opaque blobs. However, as an application developer it can be helpful to understand the format of access tokens to debug problems.

All Daml ledgers represent access tokens as JSON Web Tokens (JWTs), and there are two formats of the JSON payload used by Daml ledgers.

Note: To generate access tokens for testing purposes, you can use the jwt.io web site.

User Access Tokens

Daml ledgers that support participant *user management* also accept user access tokens. They are useful for scenarios where an application's rights change dynamically over the application's lifetime.

User access tokens do not encode rights directly like the custom Daml claims tokens explained in the following sections. Instead, user access tokens encode the participant user on whose behalf the request is issued.

When handling such requests, participant nodes look up the participant user's current rights before checking request authorization per the *table above*. Thus the rights granted to an application can be changed dynamically using the participant user management service *without* issuing new access tokens, as would be required for the custom Daml claims tokens.

User access tokens are JWTs that follow the OAuth 2.0 standard. There are two different JSON encodings: An audience-based token format that relies on the audience field to specify that it is designated for a specific Daml participant and a scope-based audience token format which relies on the scope field to designate the purpose. Both formats can be used interchangeably but if possible, use of the audience-based token format is recommend as it is compatible with a wider range of IAMs, e.g., Kubernetes does not support setting the scope field and makes the participant id mandatory which prevents misuse of a token on a different participant.

Audience-Based Tokens

```
{
   "aud": "https://daml.com/jwt/aud/participant/someParticipantId",
   "sub": "someUserId",
   "iss": "someIdpId",
   "exp": 1300819380
}
```

To interpret the above notation:

`aud` is a required field which restricts the token to participant nodes with the given ID (e.g. `someParticipantId`)
`sub` is a required field which specifies the participant user's ID
`iss` is a field which specifies the identity provider id
`exp` is an optional field which specifies the JWT expiration date (in seconds since EPOCH)

Scope-Based Tokens

```
{
   "aud": "someParticipantId",
   "sub": "someUserId",
   "exp": 1300819380,
   "iss": "someIdpId",
   "scope": "daml_ledger_api"
}
```

To interpret the above notation:

`aud` is an optional field which restricts the token to participant nodes with the given ID
`sub` is a required field which specifies the participant user's ID
`iss` is a field which specifies the identity provider id
`exp` is an optional field which specifies the JWT expiration date (in seconds since EPOCH)
`scope` is a space-separated list of OAuth 2.0 scopes that must contain the "`daml_ledger_api`" scope

Requirements for User IDs

User IDs must be non-empty strings of at most 128 characters that are either alphanumeric ASCII characters or one of the symbols @^$.!`-#+'~_|: .

Identity providers

An identity provider configuration can be thought of as a set of participant users which:

Have a defined way to verify their access tokens
Can be administered in isolation from the rest of the users on the same participant node
Have an identity provider id unique per participant node
Have a related set of parties that share the same identity provider id

A participant node always has a statically configured default identity provider configuration whose id is the empty string "". Additionally, you can configure a small number of non-default identity providers using `IdentityProviderConfigService` by supplying a non-empty identity provider id and a JWK Set URL which the participant node will use to retrieve the cryptographic data needed to verify the access tokens.

When authenticating as a user from a non-default identity provider configuration, your access tokens must contain the `iss` field whose value matches the identity provider id. In case of the default identity provider configuration, the `iss` field can be empty or omitted from the access tokens.

Custom Daml Claims Access Tokens

This format represents the *rights* granted by the access token as custom claims in the JWT's payload, like so:

```
{
   "https://daml.com/ledger-api": {
     "ledgerId": null,
     "participantId": "123e4567-e89b-12d3-a456-426614174000",
     "applicationId": null,
     "admin": true,
     "actAs": ["Alice"],
     "readAs": ["Bob"]
   },
   "exp": 1300819380
}
```

where all of the fields are optional, and if present,

`ledgerId` and `participantId` restrict the validity of the token to the given ledger or participant node
`applicationId` requires requests with this token to use that application id or not set an application id at all, which should be used to distinguish requests from different applications
`exp` is the standard JWT expiration date (in seconds since EPOCH)
`actAs`, `readAs` and (participant) `admin` encode the rights granted by this access token

The `public` right is implicitly granted to any request bearing a non-expired JWT issued by a trusted issuer with matching `ledgerId`, `participantId` and `applicationId` values.

Note: All Daml ledgers also support a deprecated legacy format of custom Daml claims access tokens whose format is equal to the above except that the custom claims are present at the same level as `exp` in the token above, instead of being nested below `"https://daml.com/ledger-api"`.

1.12.9 Explicit Contract Disclosure (Alpha)

In Daml, you must specify upfront who can view data using *observer* annotations on contracts. To change who can see the data, you would typically need to rewrite the contract (eg an asset) with a new annotation. Canton 2.7 introduces explicit contract disclosure as a feature that allows you to seamlessly delegate contract read rights to a non-stakeholder using off-ledger data distribution. This supports efficient, scalable data sharing on the ledger.

Here are some use cases that illustrate how you might benefit from explicit contract disclosure:

You want to provide proof of the price data for a stock transaction. Instead of subscribing to price updates and potentially being inundated with thousands of price updates every minute, you could serve the price data though a traditional Web 2.0 API. You can then use that API to feed only the current price back into the ledger at the time of use. You still get the same validation and security, but reduce the amount of data being transferred manyfold.
You want to run an open market on ledger. Rather than making all bids and asks explicitly visible to all marketplace users, you serve market data though standard Web 2.0 APIs. At the point of use, the available bids and asks are fed back into the transactions to get the same

activeness and correctness guarantees that would be provided had they been shared though the observer mechanism.

Toggle the `explicit-disclosure-unsafe` flag in the participant configuration as shown below to use disclosed contracts in command submission by means of explicit contract disclosure.

Note: This feature is **experimental** and **must not** be used in production environments.

```
participants {
    participant {
        ledger-api.explicit-disclosure-unsafe = true
    }
}
```

1.12.9.1 Contract Read Delegation

Contract read delegation allows a party to acquire read rights during command submission over a contract of which it is neither a stakeholder nor an informee.

As an example application where read delegation could be used, consider a simplified trade between two parties. In this example, party **Seller** owns a unit of Digital Asset `Stock` issued by the **StockExchange** party. As the issuer of the stock, **StockExchange** also publishes the stock's `PriceQuotation` as public data, which can be used for settling trades at the correct market value. The **Seller** announces an offer to sell its stock publicly by creating an `Offer` contract that can be exercised by anyone who can pay the correct market value in terms of `IOU` units.

On the other side, party **Buyer** owns an `IOU` with 10 monetary units, which it wants to use to acquire **Seller**'s stock.

The Daml templates used to model the above-mentioned trade are outlined below.

```
template IOU
  with
    issuer: Party
    owner: Party
    value: Int
  where
    signatory issuer
    observer owner

    choice IOU_Transfer: ()
      with
        target: Party
        amount: Int
      controller owner
      do
        -- Check that the transferred amount is not higher than the current IOU
↪value
        assert (value >= amount)
        create this with issuer = issuer, owner = target, value = amount
        -- No need to create a new IOU for owner if the full value is transferred
        if value == amount then pure ()
        else void $ create this with issuer = issuer, owner = owner, value =
↪value - amount
```

(continues on next page)

(continued from previous page)

```
        pure ()

template Stock
  with
    issuer: Party
    owner: Party
    stockName: Text
  where
    signatory issuer
    observer owner

    choice Stock_Transfer: ()
      with
        newOwner: Party
      controller owner
      do
        create this with owner = newOwner
        pure ()

-- Expresses the current market value of a stock issued by the issuer.
-- Not modelled in this example: the issuer ensures that only one `PriceQuotation`
-- is active at a time for a specific `stockName`.
template PriceQuotation
  with
    issuer: Party
    stockName: Text
    value: Int
  where
    signatory issuer

    -- Helper choice to allow the controller to fetch this contract without being
↪a stakeholder.
    -- By fetching this contract, the controller (i.e. `fetcher`) proves
    -- that this contract is active and represents the current market value for
↪this stock.
    nonconsuming choice PriceQuotation_Fetch: PriceQuotation
      with fetcher: Party
      controller fetcher
      do pure this

template Offer
  with
    seller: Party
    quotationProducer: Party
    offeredAssetCid: ContractId Stock
  where
    signatory seller

    choice Offer_Accept: ()
      with
        priceQuotationCid: ContractId PriceQuotation
        buyer: Party
        buyerIou: ContractId IOU
      controller buyer
      do
        priceQuotation <- exercise
```

(continues on next page)

(continued from previous page)

```
          priceQuotationCid PriceQuotation_Fetch with
            fetcher = buyer
        asset <- fetch offeredAssetCid

        -- Assert the quotation issuer and asset name
        priceQuotation.issuer === quotationProducer
        priceQuotation.stockName === asset.stockName

        _ <- exercise
          offeredAssetCid Stock_Transfer with
            newOwner = buyer

        -- Purchase the stock at the currently published fair price.
        _ <- exercise
          buyerIou IOU_Transfer with target = seller, amount = priceQuotation.
↪value
        pure ()
```

The following snippet of *Daml Script* models the setup of the trade between the parties.

```
let stockName = "Daml"

stockCid <- submit stockExchange do
  createCmd Stock with
    issuer = stockExchange
    owner = seller
    stockName = stockName

offerCid <- submit seller do
  createCmd Offer with
    seller = seller
    quotationProducer = stockExchange
    offeredAssetCid = stockCid

priceQuotationCid <- submit stockExchange do
  createCmd PriceQuotation with
    issuer = stockExchange
    stockName = stockName
    value = 3

buyerIouCid <- submit bank do
  createCmd IOU with
    issuer = bank
    owner = buyer
    value = 10
```

Settling the trade on-ledger implies that **Buyer** exercises `Offer_Accept` on the `offerCid` contract. But how can **Buyer** exercise a choice on a contract on which it is neither a stakeholder nor a prior informee? The same question applies to **Buyer**'s visibility over the `stockCid` and `priceQuotationCid` contracts.

If **Buyer** plainly exercises the choice as shown in the snippet below, the submission will fail with an error citing missing visibility rights over the involved contracts.

```
-- Command fails with missing visibility over the contracts for buyer
```

(continues on next page)

(continued from previous page)

```
_ <- submit buyer do
  exerciseCmd offerCid Offer_Accept with priceQuotationCid = priceQuotationCid,
↪buyer = buyer, buyerIou = buyerIouCid
```

Read delegation using explicit contract disclosure

With the introduction of explicit contract disclosure, **Buyer** can accept the offer from **Seller** without having seen the involved contracts on the ledger. This is possible if the contracts' stakeholders decide to *disclose* their contracts to any party desiring to execute such a trade. **Buyer** can attach the disclosed contracts to the command submission that is exercising `Offer_Accept` on **Seller**'s `offerCid`, thus bypassing the visibility restriction over the contracts.

Note: The Ledger API uses the disclosed contracts attached to command submissions for resolving contract and key activeness lookups during command interpretation. This means that usage of a disclosed contract effectively bypasses the visibility restriction of the submitting party over the respective contract. However, the authorization restrictions of the Daml model still apply: the submitted command still needs to be well authorized. The actors need to be properly authorized to execute the action, as described in *Privacy Through Authorization*.

1.12.9.2 How do stakeholders disclose contracts to submitters?

The disclosed contract's details can be fetched by the contract's stakeholder from the contract's associated *CreatedEvent*, which can be read from the Ledger API via the active contracts and transactions queries (see *Reading from the ledger*).

The stakeholder can then share the disclosed contract details to the submitter off-ledger (outside of Daml) by conventional means, such as HTTPS, SFTP, or e-mail. A *DisclosedContract* can be constructed from the fields of the same name from the original contract's `CreatedEvent`.

Note: Only contracts created starting with Canton 2.6 can be shared as disclosed contracts. Prior to this version, contracts' **CreatedEvent** does not have `ContractMetadata` populated and cannot be used as disclosed contracts.

1.12.9.3 Attaching a disclosed contract to a command submission

A disclosed contract can be attached as part of the `Command`'s *disclosed_contracts* and requires the following fields (see *DisclosedContract* for content details) to be populated from the original *CreatedEvent* (see above):

template_id - The contract's template id.
contract_id - The contract id.
arguments - The contract's create arguments. This field is a protobuf `oneof` and it allows either passing the contract's create arguments typed (as `create_arguments`) or as a byte array (as `create_arguments_blob`). Generally, clients should use the `create_arguments_blob` for convenience since they can be received as such from the stakeholder off-ledger (see above).

metadata - The contract metadata. This field can be populated as received from the stakeholder (see below).

1.12.9.4 Trading the stock with explicit disclosure

In the example above, **Buyer** does not have visibility over the `stockCid`, `priceQuotationCid` and `offerCid` contracts, so **Buyer** must provide them as disclosed contracts in the command submission exercising `Offer_Accept`. To do so, the contracts' stakeholders must fetch them from the ledger and make them available to the **Buyer**.

Note: Daml Script support for explicit disclosure is currently not implemented. The last steps of the example are modeled using raw gRPC queries.

The contracts' stakeholders issue fetch queries to the Ledger API for retrieving the associated contract payloads. For simplicity in the example, all parties reside on participant `participant` with the Ledger API running on port `5031`.

```
# Needs to be extracted via package lookup
packageId="436c13be1424a16fb69a3dda4983b94f1965ac12c66d8a6d879ad3027ea4782d"

# Needs to be extracted via party lookup
buyerId=
 ↪"Buyer::122001002fb09c069a0f4e7badf9cb1a6d7dd9097fbdb653e1278193aa5f36b9c6b3"
stockExchangeId=
 ↪"StockExchange::122001002fb09c069a0f4e7badf9cb1a6d7dd9097fbdb653e1278193aa5f36b9c6b3
 ↪"
sellerId=
 ↪"Seller::122001002fb09c069a0f4e7badf9cb1a6d7dd9097fbdb653e1278193aa5f36b9c6b3"

# StockExchange fetches the Stock contract referenced by stockCid from the ledger
 ↪by querying the Ledger API
# (here we are using the GetTransactions query)
grpcurl -plaintext -d '{"ledgerId":"participant","begin":{"absolute":
 ↪"0000000000000000"},"end":{"boundary":"LEDGER_END"},"filter":{"filters_by_party
 ↪":{"'"$stockExchangeId"'":{"inclusive":{"template_ids":[{"package_id":"'"
 ↪$packageId"'","module_name":"StockExchange","entity_name":"Stock"}]}}}},"verbose
 ↪":true}' localhost:5031 com.daml.ledger.api.v1.TransactionService/
 ↪GetTransactions
# Result: {"transactions":[{"transaction_id":
 ↪"1220073a3db0e42b536791ed24689ec587276de2cad79887e466c380c26ffda7baf1","command_
 ↪id":"e1cbb1b7-277c-4126-bde7-13b3cb158b36","effective_at":"2023-04-05T09:11:29.
 ↪062939Z","events":[{"created":{"event_id":"
 ↪#1220073a3db0e42b536791ed24689ec587276de2cad79887e466c380c26ffda7baf1:0",
 ↪"contract_id":
 ↪"00406f5cfbe495a21d576fbc4971e5d12c1ec5de972439ca0c054bbe54883de2a9ca01122064de6a454a83ce
 ↪","template_id":{"package_id":
 ↪"436c13be1424a16fb69a3dda4983b94f1965ac12c66d8a6d879ad3027ea4782d","module_name
 ↪":"StockExchange","entity_name":"Stock"},"create_arguments":{"record_id":{
 ↪"package_id":"436c13be1424a16fb69a3dda4983b94f1965ac12c66d8a6d879ad3027ea4782d",
 ↪"module_name":"StockExchange","entity_name":"Stock"},"fields":[{"label":"issuer
 ↪","value":{"party":
 ↪"StockExchange::122001002fb09c069a0f4e7badf9cb1a6d7dd9097fbdb653e1278193aa5f36b9c6b3
 ↪"}},{"label":"owner","value":{"party":
 ↪"Seller::122001002fb09c069a0f4e7badf9cb1a6d7dd9097fbdb653e1278193aa5f36b9c6b3"}}
 ↪,{"label":"stockName","value":{"text":"Daml"}}]},"witness_parties":[
 ↪"StockExchange::122001002fb09c069a0f4e7badf9cb1a6d7dd9097fbdb653e1278193aa5f36b9c6b3
 ↪"],"agreement_text":"","signatories":[
 ↪"StockExchange::122001002fb09c069a0f4e7badf9cb1a6d7dd9097fbdb653e1278193aa5f36b9c6b3
 ↪"],"observers":[
 ↪"Seller::122001002fb09c069a0f4e7badf9cb1a6d7dd9097fbdb653e1278193aa5f36b9c6b3"],
 ↪"metadata":{"created_at":"2023-04-05T09:11:29.062939Z","driver_metadata":
```

(continues on next page)

(continued from previous page)

```
# As above, StockExchange fetches the PriceQuotation referenced by priceQuotationCid
grpcurl -plaintext -d '{"ledgerId":"participant","begin":{"absolute":"0000000000000000"},"end":{"boundary":"LEDGER_END"},"filter":{"filters_by_party":{"'"$stockExchangeId"'":{"inclusive":{"template_ids":[{"package_id":"'"$packageId"'","module_name":"StockExchange","entity_name":"PriceQuotation"}]}}}},"verbose":true}' localhost:5031 com.daml.ledger.api.v1.TransactionService/GetTransactions
# Result: {"transactions":[{"transaction_id":"1220ecf0113498df1e9a4fd9aeed82b877b71cb0a8d57fdaca188294dfdeeada5eac","command_id":"433e9786-df09-4243-ad70-1d27fee05031","effective_at":"2023-04-05T09:11:29.257808Z","events":[{"created":{"event_id":"#1220ecf0113498df1e9a4fd9aeed82b877b71cb0a8d57fdaca188294dfdeeada5eac:0","contract_id":"00e0be88a38c25bc0b3b35acd6f46de92584becf99009cb512a71727fb928c90fdca01122080169e053bd955","template_id":{"package_id":"436c13be1424a16fb69a3dda4983b94f1965ac12c66d8a6d879ad3027ea4782d","module_name":"StockExchange","entity_name":"PriceQuotation"},"create_arguments":{"record_id":{"package_id":"436c13be1424a16fb69a3dda4983b94f1965ac12c66d8a6d879ad3027ea4782d","module_name":"StockExchange","entity_name":"PriceQuotation"},"fields":[{"label":"issuer","value":{"party":"StockExchange::122001002fb09c069a0f4e7badf9cb1a6d7dd9097fbdb653e1278193aa5f36b9c6b3"}},{"label":"stockName","value":{"text":"Daml"}},{"label":"value","value":{"int64":"3"}}]},"witness_parties":["StockExchange::122001002fb09c069a0f4e7badf9cb1a6d7dd9097fbdb653e1278193aa5f36b9c6b3"],"agreement_text":"","signatories":["StockExchange::122001002fb09c069a0f4e7badf9cb1a6d7dd9097fbdb653e1278193aa5f36b9c6b3"],"metadata":{"created_at":"2023-04-05T09:11:29.257808Z","driver_metadata":"CiYKJAgBEiBsywnjtj+a0Px6A2LwSV2MrOxE9QyJDM0VpgPAEGamqg=="}}}],"offset":"0000000000000000f"}]}

# As above, Seller fetches the Offer referenced by offerCid
grpcurl -plaintext -d '{"ledgerId":"participant","begin":{"absolute":"0000000000000000"},"end":{"boundary":"LEDGER_END"},"filter":{"filters_by_party":{"'"$sellerId"'":{"inclusive":{"template_ids":[{"package_id":"'"$packageId"'","module_name":"StockExchange","entity_name":"Offer"}]}}}},"verbose":true}' localhost:5031 com.daml.ledger.api.v1.TransactionService/GetTransactions
# Result: {"transactions":[{"transaction_id":"1220af12e338e39694374f8e7fc992a9361dfbe942705bdcfb29e56f5c6668713bb3","command_id":"aecbac54-5166-450c-868d-3ee912e7073c","effective_at":"2023-04-05T09:11:29.158305Z","events":[{"created":{"event_id":"#1220af12e338e39694374f8e7fc992a9361dfbe942705bdcfb29e56f5c6668713bb3:0","contract_id":"00b8355cf81045ad6212e6168380dd9ca4b7dbe9b7f0b53c595bdc0b9e60ec6789ca011220249c851ca8927e","template_id":{"package_id":"436c13be1424a16fb69a3dda4983b94f1965ac12c66d8a6d879ad3027ea4782d","module_name":"StockExchange","entity_name":"Offer"},"create_arguments":{"record_id":{"package_id":"436c13be1424a16fb69a3dda4983b94f1965ac12c66d8a6d879ad3027ea4782d","module_name":"StockExchange","entity_name":"Offer"},"fields":[{"label":"seller","value":{"party":"Seller::122001002fb09c069a0f4e7badf9cb1a6d7dd9097fbdb653e1278193aa5f36b9c6b3"}},{"label":"quotationProducer","value":{"party":"StockExchange::122001002fb09c069a0f4e7badf9cb1a6d7dd9097fbdb653e1278193aa5f36b9c6b3"}},{"label":"offeredAssetCid","value":{"contract_id":"00406f5cfbe495a21d576fbc4971e5d12c1ec5de972439ca0c054bbe54883de[illegible]454a83ce"}}]},"witness_parties":["Seller::122001002fb09c069a0f4e7badf9cb1a6d7dd9097fbdb653e1278193aa5f36b9c6b3"],"agreement_text":"","signatories":["Seller::122001002fb09c069a0f4e7badf9cb1a6d7dd9097fbdb653e1278193aa5f36b9c6b3"],"metadata":{"created_at":"2023-04-05T09:11:29.158305Z","driver_metadata":"CiYKJAgBEiBNiC/8U069Zpc7gOt3YGmmdk+TGWEZRsNukLYri+64Sg==}}}],"offset":
```

(continues on next page)

(continued from previous page)

Buyer receives these contracts from the stakeholders and adapts them to disclosed contracts (as described in *the previous section*) in a command submission that executes `Offer_Accept` on the `offerCid`. The resulting gRPC command submission, which succeeds, is shown below.

```
# Extracted from the transaction lookup query results from above
offerCid=
↪"00b8355cf81045ad6212e6168380dd9ca4b7dbe9b7f0b53c595bdc0b9e60ec6789ca011220249c851ca8927e
↪"
priceQuotationCid=
↪"00e0be88a38c25bc0b3b35acd6f46de92584becf99009cb512a71727fb928c90fdca01122080169e053bd955
↪"
stockCid=
↪"00406f5cfbe495a21d576fbc4971e5d12c1ec5de972439ca0c054bbe54883de2a9ca01122064de6a454a83ce
↪"

# The contract id of Buyer's IOU (for conciseness, not shown in this example but
↪can be extracted by the Buyer from the getTransactions queries as above)
buyerIouCid=
↪"00cd7d7b27f1b323bb55c2b0adf2aac76657079741adf6dc98a5d977338e3c92eeca011220649fd780478bb1
↪"

stockContractCreatedAt="2023-04-05T09:11:29.062939Z"
stockContractDriverMetadata="CiYKJAgBEiA5hhYAzLWLGx4dr6MO0r1xoD/AAu/
↪Xe6H56hCOzDqOlQ=="

offerContractCreatedAt="2023-04-05T09:11:29.158305Z"
offerContractDriverMetadata="CiYKJAgBEiBNiC/
↪8U069Zpc7gOt3YGmmdk+TGWEZRsNukLYri+64Sg=="

priceQuotationContractCreatedAt="2023-04-05T09:11:29.257808Z"
priceQuotationContractDriverMetadata=
↪"CiYKJAgBEiBsywnjtj+a0Px6A2LwSV2MrOxE9QyJDM0VpgPAEGamqg=="

# Buyer exercises Offer_Accept on offerCid with populating the Command.disclosed_
↪contracts field
# with the data previously shared off-ledger for offerCid, stockCid and
↪priceQuotationCid contracts
grpcurl -plaintext -d '{"commands":{"ledger_id":"participant","workflow_id":
↪"ExplicitDisclosureWorkflow","application_id":"ExplicitDisclosure","command_id":
↪"ExplicitDisclosure-command","party":"'"$buyerId"'","commands":[{"exercise":{
↪"template_id":{"package_id":"'"$packageId"'","module_name":"StockExchange",
↪"entity_name":"Offer"},"contract_id":"'"$offerCid"'","choice":"Offer_Accept",
↪"choice_argument":{"record":{"record_id":{"package_id":"'"$packageId"'","module_
↪name":"StockExchange","entity_name":"Offer_Accept"},"fields":[{"label":
↪"priceQuotationCid","value":{"contract_id":"'"$priceQuotationCid"'"}},{"label":
↪"buyer","value":{"party":"'"$buyerId"'"}},{"label":"buyerIou","value":{
↪"contract_id":"'"$buyerIouCid"'"}}]}}}}],"submission_id":"ExplicitDisclosure-
↪submission","disclosed_contracts":[{"template_id":{"package_id":"'"$packageId"'
↪","module_name":"StockExchange","entity_name":"Stock"},"contract_id":"'"
↪$stockCid"'","create_arguments":{"record_id":{"package_id":"'"$packageId"'",
↪"module_name":"StockExchange","entity_name":"Stock"},"fields":[{"label":"issuer
↪","value":{"party":"'"$stockExchangeId"'"}},{"label":"owner","value":{"party":"'
↪"$sellerId"'"}},{"label":"stockName","value":{"text":"Daml"}}]},"metadata":{
↪"created_at":"'"$stockContractCreatedAt"'","driver_metadata":"'"
↪$stockContractDriverMetadata"'"}},{"template_id":{"package_id":"'"$packageId"'",
↪"module_name":"StockExchange","entity_name":"Offer"},"contract_id":"'"$offerCid"
↪'","create_arguments":{"record_id":{"package_id":"'"$packageId"'","module_name":
↪"StockExchange","entity_name":"Offer"},"fields":[{"label":"seller","value":{
↪"party":"'"$sellerId"'"}},{"label":"quotationProducer","value":{"party":"'"
↪$stockExchangeId"'"}},{"label":"offeredAssetCid","value":{"contract_id":"'"
↪$stockCid"'"}}]},"metadata":{"created_at":"'"$offerContractCreatedAt"'","driver_
```

(continues on next page)

(continued from previous page)

1.13 Resource Management in Daml Application Design

This section discusses approaches to avoiding potential resource pitfalls by strengthening Daml contract design.

Note: In this document, we focus on building resilience into the system by writing performant Daml code at design time. We are not concerned with ledger optimization. Read more about Canton performance and scaling.

1.13.1 Managing Latency and Throughput

1.13.1.1 Problem Definition

Latency is a measure of how long a business transaction takes to complete. Throughput measures, on average, the number of business transactions possible per second while taking into account any lower or upper bounds which may point to bottlenecks. Defense against latency and throughput issues can be written into the Daml application during design.

First we need to identify the potential bottlenecks in a Daml application. We can do this by analyzing the domain-specific transactions.

Each Daml business transaction kicks off when a Ledger API client sends the commands `create` or `exercise` to a participant.

Important:

Ledger transactions are not synonymous with business transactions.
Often a complete business transaction spans multiple workflow steps and thus multiple ledger transactions.
Multiple business transactions can be processed in a single ledger transaction through batching.
Expected ledger transaction latencies are on the order of 0.5-1 seconds on database sequencers, and multiple seconds on blockchain sequencers.

Refer to the Daml execution model that describes a ledger transaction processed by the Canton ledger. The table below highlights potential resource-intensive activities at each step.

Step	Participant	Resources used	Possible bottleneck drivers
Interpretation	Submitting participant node	1. CPU 2. Memory 3. DB read access	1. Calculation complexity 2. Size and number of variables 3. Number of contract fetches
Blinding	Submitting participant node	CPU/memory	Number and size of views
Submission	Submitting participant node Sequencer	1. CPU 2. Memory	1. Serialization/deserialization 2. Transaction size/number of views
Sequencing	Sequencer	1. Backend storage 2. Network bandwidth	1. Transaction size 2. Transaction size/number of views
Validation	Receiving participant nodes	1. Network bandwidth 2. CPU 3. Memory 4. DB read throughput	1. Transaction size -> download, deserialization, storage costs 2. Computation complexity 3. Number of contract fetch reads 4. Number and size of variables
Confirmation	Validating participant nodes Sequencer	1. Network bandwidth 2. Sequencer network 3. Backend write throughput	Number of confirming parties
Mediation	Mediator nodes	1. Network throughput 2. CPU 3. Memory	Number of confirming parties
Commit	Mediator nodes Sequencer	1. CPU 2. Memory 3. DB 4. Network bandwidth	Number of confirming parties

Possible Throughput Bottlenecks in Order of Likelihood

1. Transaction size causing high serialization/deserialization and encryption/decryption costs on participant nodes.
2. Transaction size causing sequencer backend overload, especially on blockchains.
3. High interpretation and validation cost due to calculation complexity or memory use.
4. Large number of involved nodes and associated network bandwidth on sequencer.

Latency can also be affected by the above factors. However, baseline latency usually has more to do with system set-up issues (DB or blockchain latency) rather than Daml modeling problems.

Solutions

1. **Minimize transaction size.**

Each of the following actions in Daml adds a node to the transaction containing the payload of the contract being acted on. A large number of such operations, and/or operations of this kind on large contracts, are the most common cause of performance bottlenecks.

```
create
fetch
fetchByKey
lookupByKey
exercise
exerciseByKey
archive
```

Use the above actions sparingly. For example, if contracts have intermediary states within a transaction, you can often skip them by writing only the end state. For example:

```
template Incrementor
with
p : Party
n : Int
where
signatory p

choice Increment : ContractId Incrementor
    controller p
    do create this with n = n+1

-- This adds all m-1 intermediary versions of
-- the contract to the transaction tree
choice BadIncrementMany : ContractId Incrementor
    with m : Int
    controller p
    do foldlA (\self' _ -> exercise self' Increment) self [1..m]

-- This only adds the end result to the transaction
choice GoodIncrementMany : ContractId Incrementor
    with m : Int
    controller p
    do create this with n = n+m
```

When you need to read a contract, or act on a single contract in multiple ways, you can often bundle those operations into a single action. For example:

```
   template Asset
with
  issuer : Party
  owner : Party
  quantity : Decimal
where
  signatory [issuer, owner]

  -- BadMerge acts on each of the otherCids three times:
  -- Once for validation
  -- Once to extract the quantities
  -- Once to archive
  choice BadMerge : ContractId Asset
    with otherCids : [ContractId Asset]
    controller owner
    do
      -- validate the cids.
      forA_ otherCids (\cid -> do
        other <- fetch cid
        assert (other.issuer == issuer && other.owner == owner))

      -- extract the quantities
      quantities <- forA otherCids (\cid -> do
        other <- fetch cid
        return other.quantity)

      -- archive the others
      forA_ otherCids archive

      create this with quantity = quantity + sum quantities

  -- Allow us to do a fetch and an archive in one action
  choice ConsumingFetch : Asset
    controller owner
    do return this

  -- GoodMerge only acts on each of the other assets once.
  choice GoodMerge : ContractId Asset
    with otherCids : [ContractId Asset]
    controller owner
    do
      -- Get and archive the others
      others <- forA otherCids (`exercise` ConsumingFetch)

      -- validate
      forA_ others (\other -> do
        assert (other.issuer == issuer && other.owner == owner))

      -- extract the quantities
      let quantities = map (.quantity) others

      create this with quantity = quantity + sum quantities
```

Separate templates for large payloads that change rarely and require minimum access from those for fields that change with almost every action. This optimizes resource consumption for multiple

business transactions.

This batching approach makes updates in one transaction submission rather than requiring separate transactions for each update. Note: this option can cause a small increase in latency and may increase the possibility of command failure but this can be avoided. For example:

```
template T
with
p : Party
where
signatory p

choice Foo : ()
    controller p
    do return ()

batching : Script ()
batching = do
p <- allocateParty "p"

-- without batching we have 10 ledger
-- transactions.
cid1 <- submit p do createCmd T with ..
cid2 <- submit p do createCmd T with ..
cid3 <- submit p do createCmd T with ..
cid4 <- submit p do createCmd T with ..
cid5 <- submit p do createCmd T with ..

submit p do exerciseCmd cid1 Foo
submit p do exerciseCmd cid2 Foo
submit p do exerciseCmd cid3 Foo
submit p do exerciseCmd cid4 Foo
submit p do exerciseCmd cid5 Foo

-- With batching, there are only two ledger transactions.
cids <- submit p do
replicateA 5 $ createCmd T with ..
submit p do
forA_ cids (`exerciseCmd` Foo)
```

2. CPU and memory issues: Use the *Daml profiler* to analyze Daml code execution.
3. Once you feel interpretation is not the bottleneck, scale up your machine.

Tip: Profile the JVM and monitor your databases to see where the bottlenecks occur.

1.13.2 Avoid Contention Issues

Measuring the performance of business applications involves more than considering the transactions per second and transaction latency of the underlying blockchain and Distributed Ledger Technology (DLT). Blockchains are distributed systems; even the highest-performance blockchains have considerably higher transaction latencies than traditional databases. These factors make the systems prone to contention, which can stifle the performance of applications when not handled appropriately.

It is, unfortunately, easy to design low-performance applications even on a high-performance blockchain system. Applications that initially perform well may fail under pressure. It is better to plan around contention in your application design than to fix issues later. The marginal cost of including extra business logic within a blockchain transaction is often small.

Contention is expected in distributed systems. The aim is to reduce it to acceptable levels and handle it gracefully, not to eliminate it at all costs. If contention only occurs rarely, it may be cheaper for both performance and complexity to simply let the occasional allocation fail and retry, rather than implement an advanced technique to avoid it.

As an added benefit to reducing contention issues, carefully bundling or batching strategic business logic can improve performance by yielding business transaction throughput that far exceeds the blockchain transaction throughput.

1.13.2.1 Contention in Daml

Daml uses an unspent transaction output (UTXO) ledger model. UTXO models enable higher performance by supporting parallel transactions. This means that you can send new transactions while other transactions are still processing. The downside is that contention can occur if a second transaction arrives while a conflicting earlier transaction is still pending.

Daml guarantees that there can only be one consuming choice exercised per contract. If you try to commit two transactions that would consume the same contract, you have write-write contention.

Contention can also result from incomplete or stale knowledge. For example, a contract may have been archived, but a client hasn't yet been notified due to latencies or a privacy model might prevent the client from ever knowing. If you try to commit two transactions on the same contract where one transaction reads and the other one consumes an input, you run the risk of a read-write contention.

A contract is considered pending when you do not know if the output has been consumed. It is best to assume that your transactions will go through and to treat pending ones as probably consumed. You must also assume that acting on a pending contract will fail.

You need to wait while the sequencer is processing a transaction in order to confirm that an input was consumed from a consuming input request. If you do not get confirmation back from the first transaction before submitting a second transaction on the same contract, the sequence is not guaranteed. The only way to avoid this conflict is to control the sequence of those two transactions.

Ledger state is read in the following places within the Daml Execution Model :

1. A client submits a command based on the client's latest view of the state of the shared ledger. The command might include references to `ContractIds` that the client believes are active.
2. During interpretation, ledger state is used to look up active contracts.
3. During validation, ledger state is again used to look up contracts and to validate the transaction by reinterpreting it.

Contention can occur both between #1 and #2 and between #2 and #3:

The client is constructing the command in #1 based on contracts it believes to be active. But by the time the participant performs interpretation in #2, it has processed the commit of another transaction that consumed those contracts. The participant node rejects the command due to contention.
The participant successfully constructs a transaction in #2 based on contracts it believes to be active. But by the time validation happens in #3, another transaction that consumes the same contracts has already been sequenced. The validating participants reject the command due to contention.

The complete and relevant ledger state at the time of the transaction is known only after sequencing, which happens between #2 and #3. That ledger state takes precedence to ensure double spend protection.

Contention slows performance significantly. While you cannot avoid contention completely, you can design logic to minimize it. The same considerations apply to any UTXO ledger.

1.13.2.2 Reduce Contention

Contention is natural and expected when programming within a distributed system like Daml in which every action is asynchronous. It is important to understand the different causes of contention, be able to diagnose the root cause if errors of this type occur, and be able to avoid contention by designing contracts appropriately.

You can use different techniques to manage contention and to improve performance by increasing throughput and decreasing latency. These techniques include the following:

Add retry logic.
Run transactions that have causality in series.
Bundle or batch business logic to increase business transaction throughput.
Maximize parallelism with techniques such as sharding, while ensuring no contention between shards.
Split contracts across natural lines to reduce single, high-contention contracts.
Avoid write-write and write-read contention on contracts. This type of contention occurs when one requester submits a transaction with a consuming exercise on a contract while another requester submits another exercise or a fetch on the same contract. This type of contention cannot be eliminated entirely, since there will always be some latency between a client submitting a command to a participant and other clients learning of the committed transaction. Here are a few scenarios and specific measures you can take to reduce this type of collision:

- Shard data. Imagine you want to store a user directory on the ledger. At the core, this is of type `[(Text, Party)]`, where `Text` is a display name and `Party` is the associated Party. If you store this entire list on a single contract, any two users wanting to update their display name at the same time will cause a collision. If you instead keep each `(Text, Party)` on a separate contract, these write operations become independent from each other.
  A helpful analogy when structuring your data is to envision that a template defines a table, where a contract is a row in that table. Keeping large pieces of data on a contract is like storing big blobs in a database row. If these blobs can change through different actions, you have write conflicts.
- Use non-consuming choices, where possible, as they do not collide. Non-consuming choices can be used to model events that have occurred, so instead of creating a

short-lived contract to hold some data that needs to be referenced, that data could be recorded as a ledger event using a non-consuming choice.

- Avoid workflows that encourage multiple parties to simultaneously exercise a consuming choice on the same contract. For example, imagine an auction contract containing a field `highestBid : (Party, Decimal)`. If Alice tries to bid $100 at the same time that Bob tries to bid $90, it does not matter that Alice's bid is higher. The sequencer rejects the second because it has a write collision with the first transaction. It is better to record the bids in separate Bid contracts, which can be updated independently. Think about how you would structure this data in a relational database to avoid data loss due to race conditions.
- Think carefully about storing `ContractIds`. Imagine that you create a sharded user directory according to the first bullet in this list. Each user has a `User` contract that stores their display name and party. Now assume that you write a chat application, where each `Message` contract refers to the sender by `ContractId` User.
  If a user changes the display name, that reference goes stale. You either have to modify all messages that the user ever sent, or you cannot use the sender contract in Daml.
  Contract keys can be used to make this link inside Daml. If the only place you need to link `Party` to `User` is in the user interface, it might be best to not store contract references in Daml at all.

1.13.2.3 Example Application with Techniques for Reducing Contention

The example application below illustrates the relationship between blockchain and business application performance, as well as the impact of design choices. Trading, settlement, and related systems are core use cases of blockchain technology, so this example demonstrates different ways of designing such a system within a UTXO ledger model and how the design choices affect application performance.

The Example Minimal Settlement System

This section defines the requirements that the example application should fulfill, as well as how to measure its performance and where contention might occur. Assume that there are initial processes already in place to issue assets to parties. All of the concrete numbers in the example are realistic order-of-magnitude figures that are for illustrative purposes only.

Basic functional requirements for the example application

A trading system is a system that allows parties to swap assets. In this example, the parties are Alice and Bob, and the assets are shares and dollars. The basic settlement workflow could be:

1. **Proposal**: Alice offers Bob to swap one share for $1.
2. **Acceptance**: Bob agrees to the swap.
3. **Settlement**: The swap is settled atomically, meaning that at the same time Alice transfers $1 to Bob, Bob transfers one share to Alice.

Practical and security requirements for the example application

The following list adds some practical matters to complete the rough functional requirements of an example minimal trading system.

Parties can hold *asset positions* of different asset types which they control.
- An asset position consists of the type, owner, and quantity of the asset.
- An asset type is usually the combination of an on-ledger issuer and a symbol (such as currency, CUSIP, or ISIN).

Parties can transfer an asset position (or part of a position) to another party.
Parties can agree on a settlement consisting of a swap of one position for another.
Settlement happens atomically.
There are no double spends.
It is possible to constrain the *total asset position* of an owner to be non-negative. In other words, it is possible to ensure that settlements are funded. The total asset position is the sum of the quantities of all assets of a given type by that owner.

Performance measurement in the example application

Performance in the example can be measured by latency and throughput; specifically, settlement latency and settlement throughput. Another important factor in measuring performance is the ledger transaction latency.

Settlement latency: the time it takes from one party wanting to settle (just before the proposal step) to the time that party receives final confirmation that the settlement was committed (after the settlement step). For this example, assume that the best possible path occurs and that parties take zero time to make decisions.
Settlement throughput: the maximum number of settlements per second that the system as a whole can process over a long period.
Transaction latency: the time it takes from when a client application submits a command or transaction to the ledger to the time it receives the commit confirmation. The length of time depends on the command. A transaction settling a batch of 100 settlements will take longer than a transaction settling a single swap. For this example, assume that transaction latency has a simple formula of a fixed cost `fixed_tx` and a variable processing cost of `var_tx` times the number of settlements, as shown here:
`transaction latency = fixed_tx + (var_tx * #settlements)`
Note that the example application does not assign any latency cost to settlement proposals and acceptances.
For the example application, assume that:
- `fixed_tx = 250ms`
- `var_tx = 10ms`

To set a baseline performance measure for the example application, consider the simplest possible settlement workflow, consisting of one proposal transaction plus one settlement transaction done back-to-back. The following formula approximates the settlement latency of the simple workflow:

```
    (2 * fixed_tx) + var_tx
= (2 * 250ms) + 10ms
= 510ms
```

To find out how many settlements per second are possible if you perform them in series, throughput evaluates to the following formula (there are 1,000ms in one second):

```
    1000ms / (fixed_tx + var_tx) settlements per second
= 1000ms / (250ms + 10ms)
= 1000 / 260
= 3.85 or ≈ 4 settlements per second
```

These calculations set the optimal baselines for a high performance system.

The next goal is to increase throughput without dramatically increasing latency. Assume that the underlying DLT has limits on total throughput and on transaction size. Use a simple cost model in a unit called `dlt_min_tx` representing the minimum throughput unit in the DLT system. An empty transaction has a fixed cost `dlt_fixed_tx` which is:

```
    dlt_fixed_tx = 1 dlt_min_tx
```

Assume that the ratio of the marginal throughput cost of a settlement to the throughput cost of a transaction is roughly the same as the ratio of marginal latency to transaction latency (shown previously). A marginal settlement throughput cost `dlt_var_tx` can then be determined by this calculation:

```
    dlt_var_tx = ratio * dlt_fixed_tx
= dlt_var_tx = (var_tx / fixed_tx) * dlt_fixed_tx
= dlt_var_tx = 10sm/250ms * dlt_fixed_tx
= dlt_var_tx = 0.04 * dlt_fixed_tx
```

and, since from previously

```
    dlt_fixed_tx = 1 dlt_min_tx
```

then

```
    dlt_var_tx = 0.04 * dlt_min_tx
```

Even with good parallelism, ledgers have limitations. The limitations might involve CPUs, databases, or networks. Calculate and design for whatever ceiling you hit first. Specifically, there is a maximum throughput `max_throughput` (measured in `dlt_min_tx/second`) and a maximum transaction size `max_transaction` (measured in `dlt_min_tx`). For this example, assume that `max_throughput` is limited by being CPU-bound. Assume that there are 10 CPUs available and that an empty transaction takes 10ms of CPU time. For each second:

```
    max_throughput = 10 * each CPU's capacity
```

Each `dlt_min_tx` takes 10ms and there are 1,000 ms in a second. The capacity for each CPU is then 100 `dlt_min_tx` per second. The throughput calculation becomes:

```
    max_throughput = 10 * 100 dlt_min_tx/second
= max_throughput = 1,000 dlt_min_tx/second
```

Similarly, `max_transaction` could be limited by message size limit. For this example, assume that the message size limit is 3 MB and that an empty transaction `dlt_min_tx` is 1 MB. So

```
    max_transaction = 3 * dlt_min_tx
```

One of the three transactions needs to hold an approval with no settlements. That leaves the equivalent of `(2 * dlt_min_tx)` available to hold many settlements in the biggest possible transaction. Using the ratio described earlier, each marginal settlement `dlt_var_tx` takes `0.04 * dlt_min_tx`. So the maximum number of settlements per second is:

```
    (2 * dlt_min_tx)/(0.04 * dlt_min_tx)

= 50 settlements/second
```

Using the same assumptions, if you process settlements in parallel rather than in series (with only one settlement per transaction), latency stays constant while settlement throughput increases. Earlier, it was noted that a simple workflow can be `(2 * fixed_tx) + var_tx`. In the DLT system, the simple workflow calculation is:

```
    (2 * dlt_min_tx) + dlt_var_tx

= (2 * dlt_min_tx) + (0.04 * dlt_min_tx)

= 2.04 * dlt_min_tx
```

It was assumed earlier that max_throughput is `1,000 dlt_min_tx/second`. So the maximum number of settlements per second possible through parallel processing alone in the example DLT system is:

```
    1,000/2.04 settlements per second

= 490.196 or ~490 settlements per second
```

These calculations provide a baseline when comparing various techniques that can improve performance. The techniques are described in the following sections.

Prepare Transactions for Contention-Free Parallelism

This section examines which aspects of UTXO ledger models can be processed in parallel to improve performance. In UTXO ledger models, the state of the system consists of a set of immutable contracts, sometimes also called UTXOs.

Only two things can happen to a contract: it is created and later it is consumed (or spent). Each transaction is a set of input contracts and a set of output contracts, which may overlap. The transaction creates any output contracts that are not also consumed in the same transaction. It also consumes any input contracts, unless they are defined as non-consumed in the smart contract logic.

Other than smart contract logic, the execution model is the same for all UTXO ledger systems:

1. **Interpretation**: the submitting party precalculates the transaction, which consists of input and output contracts.
2. **Submission**: the submitting party submits the transaction to the network.
3. **Sequencing**: the consensus algorithm for the network assigns the transaction a place in the total order of all transactions.
4. **Validation**: the transaction is validated and considered valid if none of the inputs were already spent by a previous transaction.
5. **Commitment**: the transaction is committed.
6. **Response**: the submitting party receives a response that the transaction was committed.

The only step in this process which has a sequential component is sequencing. All other stages of transaction processing are parallelizable, which makes UTXO a good model for high-performance systems. However, the submitting party has a challenge. The interpretation step relies on knowing

possible input contracts, which are by definition unspent outputs from a previous transaction. Those outputs only become known in the response step, after a minimum delay of `fixed_tx`.

For example, if a party has a single $1,000 contract and wants to perform 1,000 settlements of $1 each, sequencing in parallel for all 1,000 settlements leads to 1,000 transactions, each trying to consume the same contract. Only one succeeds, and all the others fail due to contention. The system could retry the remaining 999 settlements, then the remaining 998, and so on, but this does not lead to a performant system. On the other hand, using the example latency of 260ms per settlement, processing these in series would take 260s or four minutes 20s, instead of the theoretical optimum of one second given by `max_throughput`. The trading party needs a better strategy. Assume that:

```
max_transaction > dlt_fixed_tx + 1,000 * dlt_var_tx = 41 dlt_min_tx
```

The trading party could perform all 1,000 settlements in a single transaction that takes:

```
fixed_tx + 1,000 * var_tx = 10.25s
```

If the latency limit is too small or this latency is unacceptable, the trading party could perform three steps to split $1,000 into:

10 * $100
100 * $10
1,000 * $1

and perform the 1,000 settlements in parallel. Latency would then be theoretically around:

```
3 * fixed_tx + (fixed_tx + var_tx) = 1.01s
```

However, since the actual settlement starts after 750 ms, and the `max_throughput` is `1,000 dlt_min_tx/s`, it would actually be:

```
0.75s + (1,000 * (dlt_fixed_tx + dlt_var_tx)) / 1,000 dlt_min_tx/s
= 1.79s
```

These strategies apply to one particular situation with a very static starting state. In a real-world high performance system, your strategy needs to perform with these assumptions:

There are constant incoming settlement requests, which you have limited ability to predict. Treat this as an infinite stream of random settlements from some distribution and maximize settlement throughput with reasonable latency.
Not all settlements are successful, due to withdrawals, rejections, and business errors.

To compare between different techniques, assume that the settlement workflow consists of the steps previously illustrated with Alice and Bob:

1. **Proposal**: proposal of the settlement
2. **Acceptance**: acceptance of the settlement
3. **Settlement**: actual settlement

These steps are usually split across two transactions by bundling the acceptance and settlement steps into one transaction. Assume that the first two steps, proposal and acceptance, are contention-free and that all contention is on settlement in the last step. Note that the cost model allocates the entire latency and throughput costs `var_tx` and `dlt_var_tx` to the settlement, so rather than discussing performant trading systems, the concern is for performant settlement systems. The following sections describe some strategies for trading under these assumptions and their tradeoffs.

Non-UTXO Alternative Ledger Models

As an alternative to a UTXO ledger model, you could use a replicated state machine ledger model, where the calculation of the transaction only happens after the sequencing.

The steps would be:

1. **Submission**: the submitting party submits a command to the network.
2. **Sequencing**: the consensus algorithm of the network assigns the command a place in the total order of all commands.
3. **Validation**: the command is evaluated to a transaction and then validated.
4. **Response**: the submitting party receives a response about the effect of the command.

Pros

This technique has a major advantage for the submitting party: no contention. The party pipes the stream of incoming transactions into a stream of commands to the ledger, and the ledger takes care of the rest.

Cons

The disadvantage of this approach is that the submitting party cannot predict the effect of the command. This makes systems vulnerable to attacks such as frontrunning and reordering.

In addition, the validation step is difficult to optimize. Command evaluation may still depend on the effects of previous commands, so it is usually done in a single-threaded manner. Transaction evaluation is at least as expensive as transaction validation. Simplifying and assuming that `var_tx` is mostly due to evaluation and validation cost, a single-threaded system would be limited to `1s / var_tx = 100` settlements per second. It could not be scaled further by adding more hardware.

Simple Strategies for UTXO Ledger Models

To attain high throughput and scalability, UTXO is the best option for a ledger model. However, you need strategies to reduce contention so that you can parallelize settlement processing.

Batch transactions sequentially

Since `(var_tx << fixed_tx)`, processing two settlements in one transaction is much cheaper than processing them in two transactions. One strategy is to batch transactions and submit one batch at a time in series.

Pros

This technique completely removes contention, just as the replicated state machine model does. It is not susceptible to reordering or frontrunning attacks.

Cons

As in the replicated state machine technique, each batch is run in a single-threaded manner. However, on top of the evaluation time, there is transaction latency. Assuming a batch size of `N < max_settlements`, the latency is:

```
fixed_tx + N * var_tx
```

and transaction throughput is:

```
N / (fixed_tx + N * var_tx)
```

As `N` goes up, this tends toward `1 / var_tx = 100`, which is the same as the throughput of replicated state machine ledgers.

In addition, there is the `max_settlements` ceiling. Assuming `max_settlements = 50`, you are limited to a throughput of `50 / 0.75 = 67` settlement transactions per second, with a latency of 750ms. Assuming that the proposal and acceptance steps add another transaction before settlement, the settlement throughput is 67 settlements per second, with a settlement latency of one second. This is better than the original four settlements per second, but far from the 490 settlements per second that is achievable with full parallelism.

Additionally, the success or failure of a whole batch of transactions is tied together. If one transaction fails in any way, all will fail, and the error handling is complex. This can be somewhat mitigated by using features such as Daml exception handling, but contention errors cannot be handled. As long as there is more than one party acting on the system and contention is possible between parties (which is usually the case), batches may fail. The larger the batch is, the more likely it is to fail, and the more costly the failure is.

Use sequential processing or batching per asset type and owner

In this technique, assume that all contention is within the asset allocation steps. Imagine that there is a single contract on the ledger that takes care of all bookkeeping, as shown in this Daml code snippet:

```
template AllAssets
  with
    -- A map from owner and type to quantity
    holdings : Map Party (Map AssetType Decimal)
  where
    signatory (keys holdings)
```

This is a typical pattern in replicated state machine ledgers, where contention does not matter. On a UTXO ledger, however, this pattern means that any two operations on assets experience contention. With this representation of assets, you cannot do better than sequential batching. There are many additional issues with this approach, including privacy and contract size.

Since you typically only need to touch one owner's asset of one type at a time and constraints such as non-negativity are also at that level, assets are usually represented by asset positions in UTXO ledgers, as shown in this Daml code snippet:

```
template
  with
    assetType : AssetType
    owner : Party
    quantity : Decimal
  where
    signatory assetType.issuer, owner
```

An asset position is a contract containing a triple (owner, asset type, and quantity). The total asset position of an asset type for an owner is the sum of the quantities for all asset positions with that owner and asset type. If the settlement transaction touches two total asset positions for the buy-side and two total asset positions for the sell-side, batching by asset type and owner does not help much.

Imagine that Alice wants to settle USD for EUR with Bob, Bob wants to settle EUR for GBP with Carol, and Carol wants to settle GBP for USD with Alice. The three settlement transactions all experience contention, so you cannot do better than sequential batching.

However, if you could ensure that each transaction only touches one total asset position, you could then apply sequential processing or batching per total asset position. This is always possible to do by decomposing the settlement step into the following:

1. **Buy-side allocation**: the buy-side splits out an asset position from their total asset position and allocates it to the settlement.
2. **Sell-side allocation**: the sell-side splits out an asset position from their total asset position and allocates it to the settlement.
3. **Settlement**: the asset positions change ownership.
4. **Buy-side merge**: the buy-side merges their new position back into the total asset position.
5. **Sell-side merge**: the sell-side merges their new position back into the total asset position.

This does not need to result in five transactions.

> Buy-side allocation is usually done as part of a settlement proposal.
> Sell-side allocation is typically handled as part of the settlement.
> Buy-side merge and sell-side merge technically do not need any action. By definition of total asset positions, merging is an optional step. It is easy to keep things organized without extra transactions. Every time a total asset position is touched as part of buy-side allocation or sell-side allocation above, you merge all positions into a single one. As long as there is a similar amount of inbound and outbound traffic on the total asset position, the number of individual positions stays low.

Pros

Assuming that a settlement is considered complete after the settlement step and that you bundle the allocation steps above into the proposal and settlement steps, the system performance will stay at the optimum settlement latency of 510ms.

Also, if there are enough open settlements on distinct total asset positions, the total throughput may reach up to the optimal 490 settlements per second.

With batch sizes of `N=50` for both proposals and settlements and sufficient total asset positions with open settlements, the maximum theoretical settlement throughput is:

```
50 stls * 1,000 dlt_min_tx/s / (2 * dlt_fixed_tx + 50 * dlt_var_tx) = 12,
500 stls/s
```

Cons

Without batching, you are limited to the original four outgoing settlements per second, per total asset position. If there are high-traffic assets, such as the USD position of a central counterparty, this can bottleneck the system as a whole.

Using higher batch sizes, you have the same tradeoffs as for sequential batching, except that it is at a total asset position level rather than a global level. Latency also scales exactly as it does for sequential batching.

Using a batch size of 50, you would get settlement latencies of around 1.5s and a maximum throughput per total asset position of 67 settlements per second, per total asset position.

Another disadvantage is that allocating the buy-side asset in a transaction before the settlement means that asset positions can be locked up for short periods.

Additionally, if the settlement fails, the already allocated asset needs to be merged back into the total asset position.

Shard Asset Positions for UTXO Ledger Models

In systems where peak loads on a single total asset position is in the tens or hundreds of settlements per second, more sophisticated strategies are needed. The total asset positions in question cannot be made up of a single asset position. They need to be sharded.

Shard total asset positions without global constraints

Consider a total asset position that represents a bookkeeping position without any on-ledger constraints. For example, the trading system may deal with fiat settlement off-ledger, and you simply want to record a balance, whether it is positive or negative. In this situation, you can easily get rid of contention altogether by assigning all allocations an arbitrary amount. To allocate $1 to a settlement, write two new asset positions of $1 and -$1 to the ledger, then use the $1 to allocate. The total asset position is unchanged.

Pros

This approach removes all contention on a total asset position.

Trading between two such total asset positions without global constraints can run at the theoretically optimal latency and throughput. Combining this with batching of batch size 50, it is possible to achieve settlements per second up to the same 12,500 settlements per second per total asset position that are possible globally.

Cons

Besides the inability to enforce any global constraints on the total asset position, this creates many new contracts. At 500 settlements per second, two allocations per settlement, and two new assets per allocation, that results in 2,000 new asset positions per second, which adds up quickly.

This effect has to be mitigated by a netting automation that nets them up into a single position once a period (for example, every time it sees >= 100 asset positions for a total position). This automation does not contend with the trading, but it adds up to 20 large transactions per second to the system and slightly reduces total throughput.

Shard total asset positions with global constraints

As an example of a global constraint, assume that the total asset position has to stay positive. This is usually done by ensuring that each individual asset position is positive. If that is the case, the strategy is to define a sharding scheme where the total position is decomposed into `N` smaller shard positions and then run sequential processing or batching per shard position.

Each asset position has to be clearly assignable to a single shard position so that there is no contention between shards. The partitioning of the total asset position does not have to be done on-ledger. If the automation for all shards can communicate off-ledger, it is possible to run a sharding strategy where you simply set the total number of desired asset positions.

For example, assume that there should be 100 asset positions for a total asset position with some minimal value.

The automation keeps track of a synchronized pending set of asset positions, which marks asset positions that are in use.
Every time the automation triggers (which may happen concurrently), it looks at how many asset positions there are relative to the desired 100 and how much quantity is needed to allocate the open settlements.
It then selects an appropriate set of non-pending asset positions so that it can allocate the open settlements and return new asset positions to move the total number closer to 100.
Before sending the transaction, it adds those positions to the pending set to make sure that another thread does not also use them.

Alternatively, if you have a sufficiently large total position compared to settlement values, you can pick the 99th percentile `p_99` of settlement values and maintain `N-1` positions of value between `p_99` and `2 * p_99` and one of the (still large) remainder. 99% of transactions will be processed in the `N-1` shard positions, and the remaining 1% will be processed against the remaining pool. Whenever a shard moves out of the desired range, it is balanced against the pool.

Pros

Assuming that there is always enough liquidity in the total asset position, the performance can be the same as without global constraints: up to 12,500 settlements per second on a single total asset position.

Cons

If settlement values are large compared to total asset holdings, this technique helps little. In an extreme case, if every settlement needs more than 50% of the total holding, it does not perform any better than the sequential processing or batching per asset type and owner technique.

In realistic scenarios where settlement values are distributed on a broad range relative to total asset position and those relativities change as holdings go up and down, developing strategies that perform optimally is complex. There are competing priorities that need to be balanced carefully:

Keeping the total number of asset positions limited so that the number of active contracts does not impact system performance.
Having sufficient large asset positions so that frequent small settlements can be processed in parallel.
Having a mechanism that ensures large settlements, possibly requiring as much as 100% of the available total asset position, are not blocked.

1.13.3 Managing Active Contract Set (ACS) Size

1.13.3.1 Problem Definition

The Active Contract Set (ACS) size makes up the load related to the number of active contracts in the system at any one time. It means the totality of all the contracts that have been created but not yet archived. ACS size may come from a deliberate Daml workflow design, but the size may also be unexpected when insufficient care is given to supporting and auxiliary contract lifetimes.

Tip: See the documentation on Daml contracts for more information.

In Daml systems, ACS size can reach orders of magnitude higher than synonymous loads in common database or blockchain systems. When the ACS size is in the high 100s GBs or TBs, local database

access performance may deteriorate. We will look at potential issues around large ACS size and possible solutions.

1.13.3.2 Relational Databases

Large ACS can have a negative impact on many aspects of system performance in relational databases. The following points focus on PostgreSQL as the underlying database; the details differ in the case of Oracle but the results are similar.

Large ACS size directly affects the resource consumption and performance of a Ledger API client application dealing with a large data set that may not fit into the memory or the application database.
ACS size directly affects the speed at which the ACS can be transmitted from the Ledger API server using the ActiveContractService. In extreme cases, it could take hours to transfer the complete set requested by the application due to the limits imposed by the gRPC channel capacity and the speed of storage queries.
Increased latency is a less direct impact which shows up wherever a query is issued to the database index to make progress. Large ACS size means that the corresponding indices are also large, and at a certain point they will no longer fit into the shared-buffer space. It then takes increasingly longer for the database engine to produce query results. This affects activities such as contract lookups during the command submission, transaction tree streaming, or pointwise transaction lookups.
Large ACS size may affect the speed at which the database underpinning the participant ingests new transactions. Normally, as new updates pour in the write-ahead log commits the table and index changes immediately. Those updates come in two shapes; full-page writes or differential writes. With large volumes, many are full-page writes.
Finally, many dirty pages also translate into prolonged and expensive flushes to the disk as part of the checkpointing process.

Solutions

Pay attention to the lifetime of the contracts. Make sure that the supporting and auxiliary contracts don't clutter the ACS and archive them as soon as it is practical to do so.
Set up a frequent pruning schedule. Be aware that pruning is only effective if there are archived contracts available for pruning. If all contracts are still active, pruning has limited success. Refer to our pruning documentation for more information.
Implement an ODS in your ledger client application to limit reliance on read access to the ACS. Do this whenever you notice that the time to initialize the application from the ACS exceeds your pain level.
Monitor database performance.

- Monitor the disk read and write activity. Look for sudden changes in the operation patterns. For instance, a sudden increase in the disk's read activity may be a sign of indices no longer fitting into the shared buffers.
- Observe the performance of the database queries. Check our monitoring documentation for query metrics that can assist. You may also consider setting up a log_min_duration_statement parameter in the PostgreSQL configuration.

Set up autovacuum on the PostgreSQL database. Note that, after pruning, a lot of dead tuples will need removing.

1.13.3.3 HTTP JSON API Service

We recommend using a relational database and dedicated compute resources to manage large ACS size when using the HTTP JSON API and refer the reader to the above considerations.

Tip: See the HTTP JSON API service documentation on managing high load in the query store and server scaling and redundancy for more information.

1.14 Upgrading and Extending Daml Applications

Database schemas tend to evolve over time. A new feature in your application might need an additional choice in one of your templates. Or a change in your data model will make you application perform better. We distinguish two kinds of changes to a Daml model:

A Daml model extension
A Daml model upgrade

An *extension* adds new templates and data structures to your model, while leaving all previously written definitions unchanged.

An *upgrade* changes previously defined data structures and templates.

Whether extension or upgrade, your new code needs to be compatible with data that is already live in a production system. The next two sections show how to extend and upgrade Daml models. The last section shows how to automate the data migration process.

1.14.1 Extending Daml Applications

Consider the following simple Daml model for carbon certificates:

```
module CarbonV1 where

template CarbonCert
  with
    issuer : Party
    owner : Party
    carbon_metric_tons : Int
  where
    signatory issuer, owner
```

It contains two templates. The above template representing a carbon compensation certificate. And a second template to create the *CarbonCert* via a *Propose-Accept workflow*.

Now we want to extend this model to add trust labels for certificates by third parties. We don't want to make any changes to the already deployed model. Changes to a Daml model will result in changed package ID's for the contained templates. This means that if a Daml model is already deployed, the modified Daml code will not be able to reference contracts instantiated with the old package. To avoid this problem, it's best to put extensions in a new package.

In our example we call the new package *carbon-label* and implement the label template like

```
module CarbonLabel where

import CarbonV1

template CarbonLabel
  with
    cert : ContractId CarbonCert
    labelOwner : Party
  where
    signatory labelOwner
```

The *CarbonLabel* template references the *CarbonCert* contract of the *carbon-1.0.0* packages by contract ID. Hence, we need to import the CarbonV1 module and add the *carbon-1.0.0* to the dependencies in the *daml.yaml* file. Because we want to be independent of the Daml SDK used for both packages, we import the *carbon-1.0.0* package as data dependency

```
name: carbon-label
version: 1.0.0
dependencies:
  - daml-prim
  - daml-stdlib
data-dependencies:
  - path/to/carbon-1.0.0.dar
```

Deploying an extension is simple: just upload the new package to the ledger with the *daml ledger upload-dar* command. In our example the ledger runs on the localhost:

```
daml ledger upload-dar --ledger-port 6865 --ledger-host localhost ./daml/dist/
↪carbon-label-1.0.0.dar
```

If instead of just extending a Daml model you want to modify an already deployed template of your Daml model, you need to perform an upgrade of your Daml application. This is the content of the next section.

1.14.2 Upgrading Daml Applications

In applications backed by a centralized database controlled by a single operator, it is possible to upgrade an application in a single step that migrates all existing data to a new data model.

As a running example, let's imagine a centralized database containing carbon offset certificates. Its operator created the database schema with

```
CREATE TABLE carbon_certs (
  carbon_metric_tons VARINT,
  owner VARCHAR NOT NULL
  issuer VARCHAR NOT NULL
)
```

The certificate has a field for the quantity of offset carbon in metric tons, an owner and an issuer.

In the next iteration of the application, the operator decides to also store and display the carbon offset method. In the centralized case, the operator can upgrade the database by executing the single SQL command

```
ALTER TABLE carbon_certs ADD carbon_offset_method VARCHAR DEFAULT "unknown"
```

This adds a new column to the `carbon_certs` table and inserts the value `unknown` for all existing entries.

While upgrading this centralized database is simple and convenient, its data entries lack any kind of signature and hence proof of authenticity. The data consumers need to trust the operator.

In contrast, Daml templates always have at least one signatory. The consequence is that the upgrade process for a Daml application needs to be different.

1.14.2.1 Daml Upgrade Overview

In a Daml application running on a distributed ledger, the signatories of a contract have agreed to one specific version of a template. Changing the definition of a template, e.g., by extending it with a new data field or choice without agreement from its signatories would completely break the authorization guarantees provided by Daml.

Therefore, Daml takes a different approach to upgrades and extensions. Rather than having a separate concept of data migration that sidesteps the fundamental guarantees provided by Daml, *upgrades are expressed as Daml contracts*. This means that the same guarantees and rules that apply to other Daml contracts also apply to upgrades.

In a Daml application, it thus makes sense to think of upgrades as an *extension of an existing application* instead of an operation that replaces existing contracts with a newer version. The existing templates stay on the ledger and can still be used. Contracts of existing templates are not automatically replaced by newer versions. However, the application is extended with new templates. Then if all signatories of a contract agree, a choice can archive the old version of a contract and create a new contract instead.

1.14.2.2 Structure Upgrade Contracts

Upgrade contracts are specific to the templates that are being upgraded. But most of them share common patterns. Here is the implementation of the above `carbon_certs` schema in Daml. We have some prescience that there will be future versions of *CarbonCert*, and so place the definition of `CarbonCert` in a module named `CarbonV1`

```
module CarbonV1 where

template CarbonCert
  with
    issuer : Party
    owner : Party
    carbon_metric_tons : Int
  where
    signatory issuer, owner
```

A *CarbonCert* has an issuer and an owner. Both are signatories. Our goal is to extend this *CarbonCert* template with a field that adds the method used to offset the carbon. We use a different name for the new template here for clarity. This is not required as templates are identified by the triple *(PackageId, ModuleName, TemplateName)*.

```
module CarbonV2 where

template CarbonCertWithMethod
  with
    issuer : Party
    owner : Party
    carbon_metric_tons : Int
    carbon_offset_method : Text
  where
    signatory issuer, owner
```

Next, we need to provide a way for the signatories to agree to a contract being upgraded. It would be possible to structure this such that issuer and owner have to agree to an upgrade for each individual *CarbonCert* contract separately. Since the template definition for all of them is the same, this is usually not necessary for most applications. Instead, we collect agreement from the signatories only once and use that to upgrade all carbon certificates.

Since there are multiple signatories involved here, we use a *Propose-Accept workflow*. First, we define an *UpgradeCarbonCertProposal* template that will be created by the issuer. This template has an *Accept* choice that the *owner* can exercise. Upon execution it will then create an *UpgradeCarbonCertAgreement*.

```
template UpgradeCarbonCertProposal
  with
    issuer : Party
    owner : Party
  where
    signatory issuer
    observer owner
    key (issuer, owner) : (Party, Party)
    maintainer key._1
    choice Accept : ContractId UpgradeCarbonCertAgreement
      controller owner
      do create UpgradeCarbonCertAgreement with ..
```

Now we can define the *UpgradeCarbonCertAgreement* template. This template has one *nonconsuming* choice that takes the contract ID of a *CarbonCert* contract, archives this *CarbonCert* contract and creates a *CarbonCertWithMethod* contract with the same issuer and owner and the *carbon_offset_method* set to `unknown`.

```
template UpgradeCarbonCertAgreement
  with
    issuer : Party
    owner : Party
  where
    signatory issuer, owner
    key (issuer, owner) : (Party, Party)
    maintainer key._1
    nonconsuming choice Upgrade : ContractId CarbonCertWithMethod
      with
        certId : ContractId CarbonCert
      controller issuer
      do cert <- fetch certId
         assert (cert.issuer == issuer)
         assert (cert.owner == owner)
         archive certId
```

(continues on next page)

(continued from previous page)

```
        create CarbonCertWithMethod with
          issuer = cert.issuer
          owner = cert.owner
          carbon_metric_tons = cert.carbon_metric_tons
          carbon_offset_method = "unknown"
```

1.14.2.3 Build and Deploy `carbon-1.0.0`

Let's see everything in action by first building and deploying `carbon-1.0.0`. After this we'll see how to deploy and upgrade to `carbon-2.0.0` containing the `CarbonCertWithMethod` template.

First we'll need a sandbox ledger to which we can deploy.

```
$ daml sandbox --port 6865
```

Now we'll setup the project for the original version of our certificate. The project contains the Daml for just the `CarbonCert` template, along with a `CarbonCertProposal` template which will allow us to issue some coins in the example below.

Here is the project config.

```
name: carbon
version: 1.0.0
dependencies:
  - daml-prim
  - daml-stdlib
  - daml-script
source: .
```

Now we can build and deploy `carbon-1.0.0`.

```
$ cd example/carbon-1.0.0
$ daml build
$ daml ledger upload-dar --port 6865
```

1.14.2.4 Create carbon-1.0.0 Certificates

Let's create some certificates!

First, we run a setup script to create 3 users `alice`, `bob` and `charlie` and corresponding parties. We write out the actual party ids to a JSON file so we can later use them in Navigator.

```
$ cd example/carbon-1.0.0
$ daml script --dar .dar/dist/carbon-1.0.0.dar --script-name Setup:setup --ledger-
↪host localhost --ledger-port 6865 --output-file parties.json
```

The resulting `parties.json` file will look similar to the following but the actual party ids will vary.

```
{
  "alice": "party-19a21501-ba87-47be-90a6-
↪692dfaefe64a::12203977cedf2d394073b4c58036e047fcc590f7f2d61d82503df431473c4277fe70
↪",
```

(continues on next page)

(continued from previous page)

```
  "bob": "party-7ecb1d67-1d20-4612-be67-
↪b5741c86204d::12203977cedf2d394073b4c58036e047fcc590f7f2d61d82503df431473c4277fe70
↪"
  "charlie": "party-fae6a574-9860-422a-9fd4-
↪7ca2f7295e41::12203977cedf2d394073b4c58036e047fcc590f7f2d61d82503df431473c4277fe70
↪"
}
```

We'll use the navigator to connect to the ledger, and create two certificates issued by Alice, and owned by Bob.

```
$ cd example/carbon-1.0.0
$ daml navigator server localhost 6865
```

We point a browser to http://localhost:4000, and follow the steps:

1. **Login as `alice`:**
    1. Select Templates tab.
    2. Create a *CarbonCertProposal* with Alice as issuer and Bob as owner and an arbitrary value for the `carbon_metric_tons` field. Note that in place of Alice and Bob, you need to use the party ids from the previously created `parties.json`.
    3. Create a 2nd proposal in the same way.
2. **Login as `bob`:**
    1. Exercise the *CarbonCertProposal_Accept* choice on both proposal contracts.

1.14.2.5 Build and Deploy `carbon-2.0.0`

Now we setup the project for the improved certificates containing the *carbon_offset_method* field. This project contains only the `CarbonCertWithMethod` template. The upgrade templates are in a third `carbon-upgrade` package. While it would be possible to include the upgrade templates in the same package, this means that the package containing the new `CarbonCertWithMethod` template depends on the previous version. With the approach taken here of keeping the upgrade templates in a separate package, the `carbon-1.0.0` package is no longer needed once we have upgraded all certificates.

It's worth stressing here that extensions always need to go into separate packages. We cannot just add the new definitions to the original project, rebuild and re-deploy. This is because the cryptographically computed package identifier would change. Consequently, it would not match the package identifier of the original `CarbonCert` contracts from `carbon-1.0.0` which are live on the ledger.

Here is the new project config:

```
name: carbon
version: 2.0.0
dependencies:
  - daml-prim
  - daml-stdlib
```

Now we can build and deploy `carbon-2.0.0`.

```
$ cd example/carbon-2.0.0
$ daml build
$ daml ledger upload-dar --port 6865
```

1.14.2.6 Build and Deploy `carbon-upgrade`

Having built and deployed `carbon-1.0.0` and `carbon-2.0.0` we are now ready to build the upgrade package `carbon-upgrade`. The project config references both `carbon-1.0.0` and `carbon-2.0.0` via the `data-dependencies` field. This allows us to import modules from the respective packages. With these imported modules we can reference templates from packages that we already uploaded to the ledger.

When following this example, `path/to/carbon-1.0.0.dar` and `path/to/carbon-2.0.0.dar` should be replaced by the relative or absolute path to the DAR file created by building the respective projects. Commonly the `carbon-1.0.0` and `carbon-2.0.0` projects would be sibling directories in the file systems, so this path would be: `../carbon-1.0.0/.daml/dist/carbon-1.0.0.dar`.

```
name: carbon-upgrade
version: 1.0.0
dependencies:
  - daml-prim
  - daml-stdlib
data-dependencies:
  - path/to/carbon-1.0.0.dar
  - path/to/carbon-2.0.0.dar
```

The Daml for the upgrade contracts imports the modules for both the new and old certificate versions.

```
module UpgradeFromCarbonCertV1 where
import CarbonV1
import CarbonV2
```

Now we can build and deploy `carbon-upgrade`. Note that uploading a DAR also uploads its dependencies so if `carbon-1.0.0` and `carbon-2.0.0` had not already been deployed before, they would be deployed as part of deploying `carbon-upgrade`.

```
$ cd example/carbon-upgrade
$ daml build
$ daml ledger upload-dar --port 6865
```

1.14.2.7 Upgrade Existing Certificates from carbon-1.0.0 to carbon-2.0.0

We start the navigator again.

```
$ cd example/carbon-upgrade
$ daml navigator server localhost 6865
```

Finally, we point a browser to http://localhost:4000 and can start the carbon certificates upgrades:

1. **Login as `alice`**
    1. Select Templates tab.
    2. Create an `UpgradeCarbonCertProposal` with Alice as issuer and Bob as owner. As before, in place of Alice and Bob use the party ids from `parties.json`.
2. **Login as `bob`**
    1. Exercise the `Accept` choice of the upgrade proposal, creating an `UpgradeCarbonCertAgreement`.
3. **Login again as `alice`**

1. Use the `UpgradeCarbonCertAgreement` repeatedly to upgrade any certificate for which Alice is issuer and Bob is owner.

1.14.2.8 Further Steps

For the upgrade of our carbon certificate model above, we performed all steps manually via Navigator. However, if Alice had issued millions of carbon certificates, performing all upgrading steps manually becomes infeasible. It thus becomes necessary to automate these steps. We will go through a potential implementation of an automated upgrade in the *next section*.

1.14.3 Automating the Upgrade Process

In this section, we are going to automate the upgrade of our carbon certificate process using *Daml Script* and *Daml Triggers*. Note that automation for upgrades is specific to an individual application, just like the upgrade models. Nevertheless, we have found that the pattern shown here occurs frequently.

1.14.3.1 Structure the Upgrade

There are three kinds of actions performed during the upgrade:

1. Alice creates `UpgradeCarbonCertProposal` contracts. We assume here, that Alice wants to upgrade all `CarbonCert` contracts she has issued. Since the `UpgradeCarbonCertProposal` proposal is specific to each owner, Alice has to create one `UpgradeCarbonCertProposal` per owner. There can be potentially many owners but this step only has to be performed once assuming Alice will not issue more `CarbonCert` contracts after this point.
2. Bob and other owners accept the `UpgradeCarbonCertProposal`. To keep this example simple, we assume that there are only carbon certificates issued by Alice. Therefore, each owner has to accept at most one proposal.
3. As owners accept upgrade proposals, Alice has to upgrade each certificate. This means that she has to execute the upgrade choice once for each certificate. Owners will not all accept the upgrade at the same time and some might never accept it. Therefore, this should be a long-running process that upgrades all carbon certificates of a given owner as soon as they accept the upgrade.

Given those constraints, we are going to use the following tools for the upgrade:

1. A Daml script that will be executed once by Alice and creates an `UpgradeCarbonCertProposal` contract for each owner.
2. Navigator to accept the `UpgradeCarbonCertProposal` as Bob. While we could also use a Daml script to accept the proposal, this step will often be exposed as part of a web UI so doing it interactively in Navigator resembles that workflow more closely.
3. A long-running Daml trigger that upgrades all `CarbonCert` contracts for which there is a corresponding `UpgradeCarbonCertAgreement`.

1.14.3.2 Implementation of the Daml Script

In our Daml Script, we are first going to query the ACS (Active Contract Set) to find all `CarbonCert` contracts issued by us. Next, we are going to extract the owner of each of those contracts and remove any duplicates coming from multiple certificates issued to the same owner. Finally, we iterate over the owners and create an `UpgradeCarbonCertAgreement` contract for each owner.

```
initiateUpgrade : Setup.Parties -> Script ()
initiateUpgrade Setup.Parties{alice} = do
  certs <- query @CarbonCert alice
  let myCerts = filter (\(_cid, c) -> c.issuer == alice) certs
  let owners = dedup $ map (\(_cid, c) -> c.owner) myCerts
  forA_ owners $ \owner -> do
    debugRaw ("Creating upgrade proposal for: " <> show owner)
    submit alice $ createCmd (UpgradeCarbonCertProposal alice owner)
```

1.14.3.3 Implementation of the Daml Trigger

Our trigger does not need any custom user state and no heartbeat so the only interesting field in its definition is the rule.

```
upgradeTrigger : Trigger ()
upgradeTrigger = Trigger with
  initialize = pure ()
  updateState = \_msg -> pure ()
  registeredTemplates = AllInDar
  heartbeat = None
  rule = triggerRule
```

In our rule, we first filter out all agreements and certificates issued by us. Next, we iterate over all agreements. For each agreement we filter the certificates by the owner of the agreement and finally upgrade the certificate by exercising the `Upgrade` choice. We mark the certificate as pending which temporarily removes it from the ACS and therefore stops the trigger from trying to upgrade the same certificate multiple times if the rule is triggered in quick succession.

```
triggerRule : Party -> TriggerA () ()
triggerRule issuer = do
  agreements <-
    filter (\(_cid, agreement) -> agreement.issuer == issuer) <$>
    query @UpgradeCarbonCertAgreement
  allCerts <-
    filter (\(_cid, cert) -> cert.issuer == issuer) <$>
    query @CarbonCert
  forA_ agreements $ \(agreementCid, agreement) -> do
    let certsForOwner = filter (\(_cid, cert) -> cert.owner == agreement.owner)
↪allCerts
    forA_ certsForOwner $ \(certCid, _) ->
      emitCommands
        [exerciseCmd agreementCid (Upgrade certCid)]
        [toAnyContractId certCid]
```

The trigger is a long-running process and the rule will be executed whenever the state of the ledger changes. So whenever an owner accepts an upgrade proposal, the trigger will run the rule and upgrade all certificates of that owner.

1.14.3.4 Deploy and Execute the Upgrade

Now that we defined our Daml script and our trigger, it is time to use them! If you still have Sandbox running from the previous section, stop it to clear out all data before continuing.

First, we start sandbox passing in the `carbon-upgrade` DAR. Since a DAR includes all transitive dependencies, this includes `carbon-1.0.0` and `carbon-2.0.0`.

```
$ cd example/carbon-upgrade
$ daml sandbox --dar .daml/dist/carbon-upgrade-1.0.0.dar
```

To simplify the setup here, we use a Daml script to create 3 parties Alice, Bob and Charlie and two `CarbonCert` contracts issues by Alice, one owned by Bob and one owned by Charlie. This Daml script reuses the `Setup.setup` Daml script from the previous section to create the parties & users.

```
setup : Script Setup.Parties
setup = do
  parties@Setup.Parties{..} <- Setup.setup
  bobProposal <- submit alice $ createCmd (CarbonCertProposal alice bob 10)
  submit bob $ exerciseCmd bobProposal CarbonCertProposal_Accept
  charlieProposal <- submit alice $ createCmd (CarbonCertProposal alice charlie 5)
  submit charlie $ exerciseCmd charlieProposal CarbonCertProposal_Accept
  pure parties
```

Run the script as follows:

```
$ cd example/carbon-initiate-upgrade
$ daml build
$ daml script --dar=.daml/dist/carbon-initiate-upgrade-1.0.0.dar --script-
↪name=InitiateUpgrade:setup --ledger-host=localhost --ledger-port=6865 --output-
↪file parties.json
```

As before, `parties.json` contains the actual party ids we can use later.

If you now start Navigator from the `carbon-initiate-upgrade` directory and log in as `alice`, you can see the two `CarbonCert` contracts.

Next, we run the trigger for Alice. The trigger will keep running throughout the rest of this example.

```
$ cd example/carbon-upgrade-trigger
$ daml build
$ daml trigger --dar=.daml/dist/carbon-upgrade-trigger-1.0.0.dar --trigger-
↪name=UpgradeTrigger:upgradeTrigger --ledger-host=localhost --ledger-port=6865 --
↪ledger-user=alice
```

With the trigger running, we can now run the script to create the `UpgradeCarbonCertProposal` contracts (we could also have done that before starting the trigger). The script takes an argument of type `Parties` corresponding to the result of the previous `setup` script. We can pass this in via the `--input-file` argument.

```
$ cd example/carbon-initiate-upgrade
$ daml build
$ daml script --dar=.daml/dist/carbon-initiate-upgrade-1.0.0.dar --script-
↪name=InitiateUpgrade:initiateUpgrade --ledger-host=localhost --ledger-port=6865
↪--input-file=parties.json
```

At this point, our trigger is running and the `UpgradeCarbonCertProposal` contracts for Bob and Charlie have been created. What is left to do is to accept the proposals. Our trigger will then automatically pick them up and upgrade the `CarbonCert` contracts.

First, start Navigator and log in as `bob`. Click on the `UpgradeCarbonCertProposal` and accept it. If you now go back to the contracts tab, you can see that the `CarbonCert` contract has been archived and instead there is a new `CarbonCertWithMethod` upgrade. Our trigger has successfully upgraded the `CarbonCert`!

Next, log in as `charlie` and accept the `UpgradeCarbonCertProposal`. Just like for Bob, you can see that the `CarbonCert` contract has been archived and instead there is a new `CarbonCertWithMethod` contract.

Since we upgraded all `CarbonCert` contracts issued by Alice, we can now stop the trigger and declare the update successful.

1.15 Developer Tools

1.15.1 Daml Assistant (`daml`)

`daml` is a command-line tool that does a lot of useful things related to the SDK. Using `daml`, you can:

Create new Daml projects: `daml new <path to create project in>`
Create a new project based on the `create-daml-app` template: `daml new --template=create-daml-app <path to create project in>`
Initialize a Daml project: `daml init`
Compile a Daml project: `daml build`
This builds the Daml project according to the project config file `daml.yaml` (see *Configuration files* below).
In particular, it will download and install the specified version of the Daml SDK (the `sdk-version` field in `daml.yaml`) if missing, and use that SDK version to resolve dependencies and compile the Daml project.
Launch the tools in the SDK:

- Launch *Daml Studio*: `daml studio`
- Launch *Sandbox*, *Navigator* and the *HTTP JSON API Service*: `daml start` You can disable the HTTP JSON API by passing `--json-api-port none` to `daml start`. To specify additional options for sandbox/navigator/the HTTP JSON API you can use `--sandbox-option=opt`, `--navigator-option=opt` and `--json-api-option=opt`.
- Launch Sandbox: `daml sandbox`
- Launch Navigator: `daml navigator`
- Launch the *HTTP JSON API Service*: `daml json-api`
- Run *Daml codegen*: `daml codegen`

Install new SDK versions manually: `daml install <version>`
Note that you need to update your *project config file <#configuration-files>* to use the new version.

1.15.1.1 Full Help for Commands

To see information about any command, run it with `--help`.

1.15.1.2 Configuration Files

The Daml assistant and the SDK are configured using two files:

The global config file, one per installation, which controls some options regarding SDK installation and updates
The project config file, one per Daml project, which controls how the SDK builds and interacts with the project

Global Config File (`daml-config.yaml`)

The global config file `daml-config.yaml` is in the `daml` home directory (`~/.daml` on Linux and Mac, `C:/Users/<user>/AppData/Roaming/daml` on Windows). It controls options related to SDK version installation and upgrades.

By default it's blank, and you usually won't need to edit it. It recognizes the following options:

`auto-install`: whether `daml` automatically installs a missing SDK version when it is required (defaults to `true`)
`update-check`: how often `daml` will check for new versions of the SDK, in seconds (default to `86400`, i.e. once a day)
This setting is only used to inform you when an update is available.
Set `update-check: <number>` to check for new versions every N seconds. Set `update-check: never` to never check for new versions.
`artifactory-api-key`: If you have a license for Daml EE, you can use this to specify the Artifactory API key displayed in your user profile. The assistant will use this to download the EE edition.

Here is an example `daml-config.yaml`:

```
auto-install: true
update-check: 86400
```

Project Config File (`daml.yaml`)

The project config file `daml.yaml` must be in the root of your Daml project directory. It controls how the Daml project is built and how tools like Sandbox and Navigator interact with it.

The existence of a `daml.yaml` file is what tells `daml` that this directory contains a Daml project, and lets you use project-aware commands like `daml build` and `daml start`.

`daml init` creates a `daml.yaml` in an existing folder, so `daml` knows it's a project folder.

`daml new` creates a skeleton application in a new project folder, which includes a config file. For example, `daml new my_project` creates a new folder `my_project` with a project config file `daml.yaml` like this:

```
sdk-version: __VERSION__
name: __PROJECT_NAME__
source: daml
init-script: Main:setup
parties:
  - Alice
  - Bob
version: 1.0.0
exposed-modules:
  - Main
dependencies:
  - daml-prim
  - daml-stdlib
script-service:
  grpc-max-message-size: 134217728
  grpc-timeout: 60
  jvm-options: []
build-options: ["--ghc-option", "-Werror",
                "--ghc-option", "-v"]
```

Here is what each field means:

`sdk-version`: the SDK version that this project uses.
The assistant automatically downloads and installs this version if needed (see the `auto-install` setting in the global config). We recommend keeping this up to date with the latest stable release of the SDK. It is possible to override the version without modifying the `daml.yaml` file by setting the `DAML_SDK_VERSION` environment variable. This is mainly useful when you are working with an external project that you want to build with a specific version.
The assistant will warn you when it is time to update this setting (see the `update-check` setting in the global config to control how often it checks, or to disable this check entirely).

`name`: the name of the project. This determines the filename of the `.dar` file compiled by `daml build`.

`source`: the root folder of your Daml source code files relative to the project root.

`init-script`: the name of the Daml script to run when using `daml start`.

`parties`: the parties to display in the Navigator when using `daml start`.

`version`: the project version.

`exposed-modules`: the Daml modules that are exposed by this project, which can be imported in other projects. If this field is not specified all modules in the project are exposed.

`dependencies`: library-dependencies of this project. See *Reference: Daml Packages*.

`data-dependencies`: Cross-SDK dependencies of this project See *Reference: Daml Packages*.

`module-prefixes`: Prefixes for all modules in package See *Reference: Daml Packages*.

`script-service`: settings for the script service

- `grpc-max-message-size`: This option controls the maximum size of gRPC messages. If unspecified this defaults to 128MB (134217728 bytes). Unless you get errors, there should be no reason to modify this.
- `grpc-timeout`: This option controls the timeout used for communicating with the script service. If unspecified this defaults to 60s. Unless you get errors, there should be no reason to modify this.
- `jvm-options`: A list of options passed to the JVM when starting the script service. This can be used to limit maximum heap size via the `-Xmx` flag.

`build-options`: a list of tokens that will be appended to some invocations of `damlc` (currently

build and *ide*). Note that there is no further shell parsing applied.
`sandbox-options`: a list of options that will be passed to Sandbox in `daml start`.
`navigator-options`: a list of options that will be passed to Navigator in `daml start`.
`json-api-options`: a list of options that will be passed to the HTTP JSON API in `daml start`.
`script-options`: a list of options that will be passed to the Daml script runner when running the `init-script` as part of `daml start`.
`start-navigator`: Controls whether navigator is started as part of `daml start`. Defaults to `true`. If this is specified as a CLI argument, say `daml start --start-navigator=true`, the CLI argument takes precedence over the value in `daml.yaml`.

Recommended `build-options`

The default set of warnings enabled by the Daml compiler is fairly conservative. When you are just starting out, seeing a huge set of warnings can easily be overwhelming and distract from what you are actually working on. However, as you get more experienced and more people work on a Daml project, enabling additional warnings (and enforcing their absence in CI) can be useful.

Here are `build-options` you might declare in a project's `daml.yaml` for a stricter set of warnings.

```
build-options:
  - --ghc-option=-Wunused-top-binds
  - --ghc-option=-Wunused-matches
  - --ghc-option=-Wunused-do-bind
  - --ghc-option=-Wincomplete-uni-patterns
  - --ghc-option=-Wredundant-constraints
  - --ghc-option=-Wmissing-signatures
  - --ghc-option=-Werror
```

Each option enables a particular warning, except for the last one, `-Werror`, which turns every warning into an error; this is especially useful for CI build arrangements. Simply remove or comment out any line to disable that category of warning. See the Daml forum for a discussion of the meaning of these warnings and pointers to other available warnings.

1.15.1.3 Build Daml Projects

To compile your Daml source code into a Daml archive (a `.dar` file), run:

```
daml build
```

You can control the build by changing your project's `daml.yaml`:

`sdk-version` The SDK version to use for building the project.
`name` The name of the project.
`source` The path to the source code.

The generated `.dar` file is created in `.daml/dist/${name}.dar` by default. To override the default location, pass the `-o` argument to `daml build`:

```
daml build -o path/to/darfile.dar
```

1.15.1.4 Manage Releases

You can manage SDK versions manually by using `daml install`.

To download and install SDK of the latest stable Daml version:

```
daml install latest
```

To download and install the latest snapshot release:

```
daml install latest --snapshots=yes
```

Please note that snapshot releases are not intended for production usage.

To install the SDK version specified in the project config, run:

```
daml install project
```

To install a specific SDK version, for example version `2.0.0`, run:

```
daml install 2.0.0
```

Rarely, you might need to install an SDK release from a downloaded SDK release tarball. **This is an advanced feature**: you should only ever perform this on an SDK release tarball that is released through the official `digital-asset/daml` github repository. Otherwise your `daml` installation may become inconsistent with everyone else's. To do this, run:

```
daml install path-to-tarball.tar.gz
```

By default, `daml install` will update the assistant if the version being installed is newer. You can force the assistant to be updated with `--install-assistant=yes` and prevent the assistant from being updated with `--install-assistant=no`.

See `daml install --help` for a full list of options.

1.15.1.5 Terminal Command Completion

The `daml` assistant comes with support for `bash` and `zsh` completions. These will be installed automatically on Linux and Mac when you install or upgrade the Daml assistant.

If you use the `bash` shell, and your `bash` supports completions, you can use the TAB key to complete many `daml` commands, such as `daml install` and `daml version`.

For `Zsh` you first need to add `~/.daml/zsh` to your `$fpath`, e.g., by adding the following to the beginning of your `~/.zshrc` before you call `compinit`: `fpath=(~/.daml/zsh $fpath)`

You can override whether bash completions are installed for `daml` by passing `--bash-completions=yes` or `--bash-completions=no` to `daml install`.

1.15.1.6 Run Commands Outside of the Project Directory

In some cases, it can be convenient to run a command in a project without having to change directories. For that use case, you can set the `DAML_PROJECT` environment variable to the path to the project:

```
DAML_PROJECT=/path/to/my/project daml build
```

Note that while some commands, most notably, `daml build`, accept a `--project-root` option, it can end up choosing the wrong SDK version so you should prefer the environment variable instead.

1.15.2 Canton Console

1.15.2.1 Introduction

Canton offers a console where you can run administrative or debugging commands.

When you run the Sandbox using `daml start` or `daml sandbox`, you are effectively starting an in-memory instance of Canton with a single domain and a single participant.

As such, you can interact with the running Sandbox using the console, just like you would in a production environment.

The purpose of this page is to give a few pointers on how the console can be used to interact with a running Sandbox. For an in-depth guide on how to use this tool against a production, staging or testing environment, *consult the main documentation for the Canton console*.

1.15.2.2 Run the Canton Console Against the Sandbox

Once you have a Sandbox running locally (for example after running `daml start` or `daml sandbox`) you can start the console with the following command (in a separate terminal):

```
daml canton-console
```

Once the console starts (it might take some time the first time) you can quit the session by running the `exit` command.

1.15.2.3 Built-in Documentation

The Canton console comes with built-in documentation. You can use the `help` command to get online documentation for top-level commands. Many objects in the console also have further built-in help that you can access by invoking the `help` method on them.

For example, you can ask for help on the `health` object by typing:

```
health.help
```

Or go more in depth about specific items within that object as in the following example:

```
health.help("status")
```

1.15.2.4 Interact With the Sandbox

One of the objects available in the Canton console represents the Sandbox itself. The object is called `sandbox` and you can use it to interact with the Sandbox. For example, you can list the DARs loaded on the Sandbox by running the following command:

```
sandbox.dars.list()
```

Among the various features available as part of the console, you can manage parties and packages, check the health of the Sandbox, perform pruning operations and more. Consult the built-in documentation mentioned above and *the main documentation for the Canton console* to learn about further capabilities.

1.15.3 Deploy to a Generic Daml Ledger

Daml ledgers expose a unified administration API. This means that deploying to a Daml ledger is no different from deploying to your local sandbox.

To deploy to a Daml ledger, run the following command from within your Daml project:

```
$ daml deploy --host=<HOST> --port=<PORT> --access-token-file=<TOKEN-FILE>
```

where `<HOST>` and `<PORT>` is the hostname and port your ledger is listening on, which defaults to port `6564`. The `<TOKEN-FILE>` is needed if your sandbox runs with *authorization* and needs to contain a JWT token for a user with an `admin` claim. If your sandbox is not set up to use any authentication it can be omitted.

Instead of passing `--host`, `--port` and `--access-token-file` flags to the command above, you can add the following section to the project's `daml.yaml` file:

```
ledger:
    host: <HOSTNAME>
    port: <PORT>
    access-token-file: <PATH TO ACCESS TOKEN FILE>
```

The `daml deploy` command will:

1. upload the project's compiled DAR file to the ledger. This will make the Daml templates defined in the current project available to the API users of the sandbox.
2. allocate the parties specified in the project's `daml.yaml` on the ledger if they are missing.

For additional interactions with the ledger, use the `daml ledger` command. Try running `daml ledger --help` to get a list of available ledger commands:

```
$ daml ledger --help
Usage: daml ledger COMMAND
  Interact with a remote Daml ledger. You can specify the ledger in daml.yaml
  with the ledger.host and ledger.port options, or you can pass the --host and
  --port flags to each command below. If the ledger is authenticated, you should
  pass the name of the file containing the token using the --access-token-file
  flag or the `daml.access-token-file` field in daml.yaml.

Available options:
  -h,--help                Show this help text
```

(continues on next page)

(continued from previous page)

```
Available commands:
  list-parties              List parties known to ledger
  allocate-parties          Allocate parties on ledger if they don't exist
  upload-dar                Upload DAR file to ledger
  fetch-dar                 Fetch DAR from ledger into file
  metering-report           Report on Ledger Use
```

1.15.3.1 Connect via TLS

To connect to the ledger via TLS, pass `--tls` to the various commands. If your ledger supports or requires mutual authentication you can pass your client key and certificate chain files via `--pem client_key.pem --crt client.crt`. Finally, you can use a custom certificate authority for validating the server certificate by passing `--cacrt server.crt`. If `--pem`, `--crt` or `--cacrt` are specified TLS is enabled automatically so `--tls` is redundant.

1.15.3.2 Configure Request Timeouts

You can configure the timeout used on API requests by passing `--timeout=N` to the various `daml ledger` commands and `daml deploy` which will set the timeout to N seconds. Note that this is a per-request timeout not a timeout for the whole command. That matters for commands like `daml deploy` that consist of multiple requests.

1.15.4 Daml REPL

The Daml REPL allows you to use the *Daml Script* API interactively. This is useful for debugging and for interactively inspecting and manipulating a ledger.

1.15.4.1 Usage

First create a new project based on the `script-example` template. Take a look at the documentation for *Daml Script* for details on this template.

```
daml new script-example --template script-example # create a project called
↪script-example based on the template
cd script-example # switch to the new project
```

Now, build the project and start *Daml Sandbox*, the in-memory ledger included in the SDK. Note that we are starting Sandbox in wallclock mode. Static time is not supported in `daml repl`.

```
daml build
daml sandbox --wall-clock-time --port=6865 --dar .daml/dist/script-example-0.0.1.
↪dar
```

Now that the ledger has been started, you can launch the REPL in a separate terminal using the following command.

```
daml repl --ledger-host=localhost --ledger-port=6865 .daml/dist/script-example-0.
↪0.1.dar --import script-example
```

The `--ledger-host` and `--ledger-port` parameters point to the host and port your ledger is running on. In addition to that, you also need to pass in the name of a DAR containing the templates and other definitions that will be accessible in the REPL. We also specify that we want to import all modules from the `script-example` package. If your modules provide colliding definitions you can also import modules individually from within the REPL. Note that you can also specify multiple DARs and they will all be available.

You should now see a prompt looking like

```
daml>
```

You can think of this prompt like a line in a `do`-block of the `Script` action. Each line of input has to have one of the following two forms:

1. An expression `expr` of type `Script a` for some type `a`. This will execute the script and print the result if `a` is an instance of `Show` and not `()`.
2. A pure expression `expr` of type `a` for some type `a` where `a` is an instance of `Show`. This will evaluate `expr` and print the result. If you are only interest in pure expressions you can also use Daml REPL *without connecting to a ledger*.
3. A binding of the form `pat <- expr` where `pat` is pattern, e.g., a variable name `x` to bind the result to and `expr` is an expression of type `Script a`. This will execute the script and match the result against the pattern `pat` bindings the matches to the variables in the pattern. You can then use those variables on subsequent lines.
4. A `let` binding of the form `let pat = y`, where `pat` is a pattern and `y` is a pure expression or `let f x = y` to define a function. The bound variables can be used on subsequent lines.
5. Next to Daml code the REPL also understands REPL commands which are prefixed by `:`. Enter `:help` to see a list of supported REPL commands.

First create two parties: A party with the display name `"Alice"` and the party id `"alice"` and a party with the display name `"Bob"` and the party id `"bob"`.

```
daml> alice <- allocatePartyWithHint "Alice" (PartyIdHint "alice")
daml> bob <- allocatePartyWithHint "Bob" (PartyIdHint "bob")
```

Next, create a `CoinProposal` from `Alice` to `Bob`

```
daml> submit alice (createCmd (CoinProposal (Coin alice bob)))
```

As Bob, you can now get the list of active `CoinProposal` contracts using the `query` function. The `debug : Show a => a -> Script ()` function can be used to print values.

```
daml> proposals <- query @CoinProposal bob
daml> debug proposals
[Daml.Script:39]: [(<contract-id>,CoinProposal {coin = Coin {issuer = 'alice',
↪owner = 'bob'}})]
```

Finally, accept all proposals using the `forA` function to iterate over them.

```
daml> forA proposals $ \(contractId, _) -> submit bob (exerciseCmd contractId
↪Accept)
```

Using the `query` function we can now verify that there is one `Coin` and no `CoinProposal`:

```
daml> coins <- query @Coin bob
daml> debug coins
```

(continues on next page)

(continued from previous page)

```
[Daml.Script:39]: [(<contract-id>,Coin {issuer = 'alice', owner = 'bob'})]
daml> proposals <- query @CoinProposal bob
[Daml.Script:39]: []
```

To exit `daml repl` press `Control-D`.

1.15.4.2 What Is in Scope at the Prompt?

In the prompt, all modules from DALFs specified in `--import` are imported automatically. In addition to that, the `Daml.Script` module is also imported and gives you access to the Daml Script API.

You can use the commands `:module + ModA ModB ...` to import additional modules and `:module - ModA ModB ...` to remove previously added imports. Modules can also be imported using regular import declarations instead of `module +`. The command `:show imports` lists the currently active imports.

```
daml> import DA.Time
daml> debug (days 1)
```

1.15.4.3 Using Daml REPL Without a Ledger

If you are only interested in pure expressions, e.g., because you want to test how some function behaves you can omit the `--ledger-host` and `-ledger-port` parameters. Daml REPL will work as usual but any attempts to call Daml Script APIs that interact with the ledger, e.g., `submit` will result in the following error:

```
daml> java.lang.RuntimeException: No default participant
```

1.15.4.4 Connecting via TLS

You can connect to a ledger that requires TLS by passing `--tls`. A custom root certificate used for validating the server certificate can be set via `--cacrt`. Finally, you can also enable client authentication by passing `--pem client.key --crt client.crt`. If `--cacrt` or `--pem` and `--crt` are passed TLS is automatically enabled so `--tls` is redundant.

1.15.4.5 Connection to a Ledger With Authorization

If your ledger requires an authorization token you can pass it via `--access-token-file`.

1.15.4.6 Using Daml REPL to Convert to JSON

Using the `:json` command you can encode serializable Daml expressions as JSON. For example using the definitions and imports from above:

```
daml> :json days 1
{"microseconds":86400000000}
daml> :json map snd coins
[{"issuer":"alice","owner":"bob"}]
```

1.15.5 Daml Studio

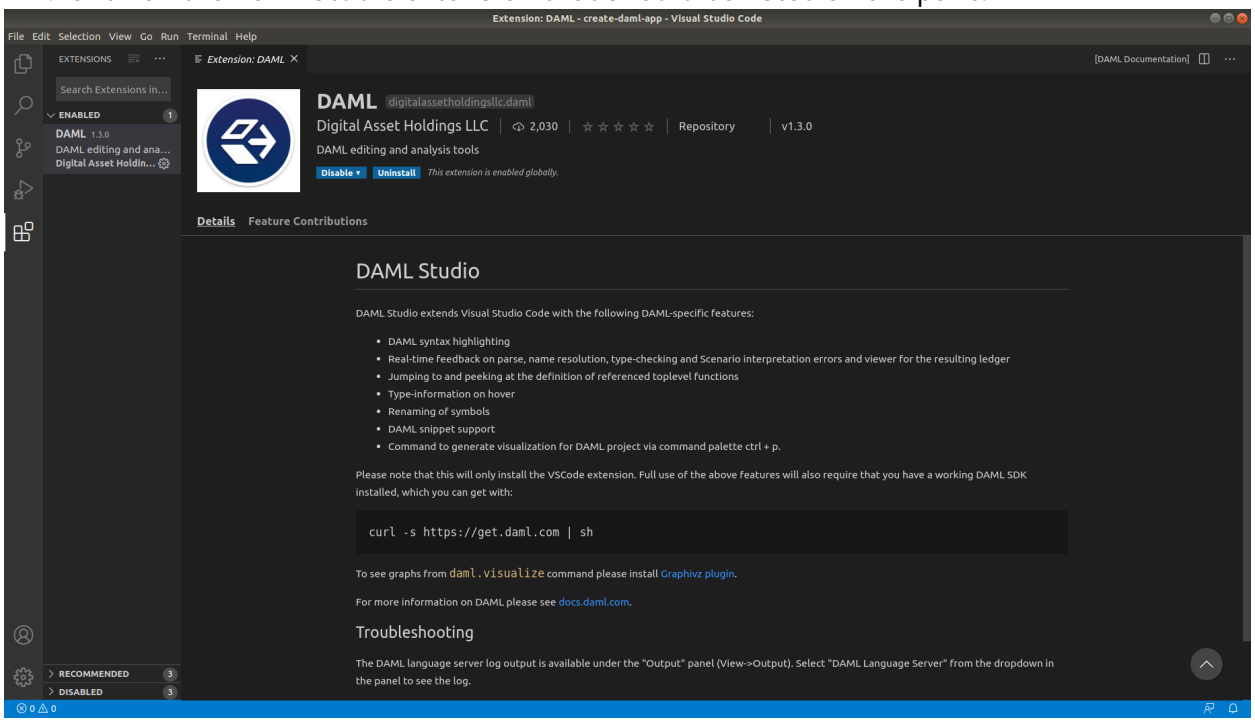
Daml Studio is an integrated development environment (IDE) for Daml. It is an extension on top of Visual Studio Code (VS Code), a cross-platform, open-source editor providing a rich code editing experience.

1.15.5.1 Install

Daml Studio is included in *the Daml SDK*.

1.15.5.2 Create Your First Daml File

1. Start Daml Studio by running `daml studio` in the current project.
   This command starts Visual Studio Code and (if needs be) installs the Daml Studio extension, or upgrades it to the latest version.
2. Make sure the Daml Studio extension is installed:
   1. Click on the Extensions icon at the bottom of the VS Code sidebar.
   2. Click on the Daml Studio extension that should be listed on the pane.

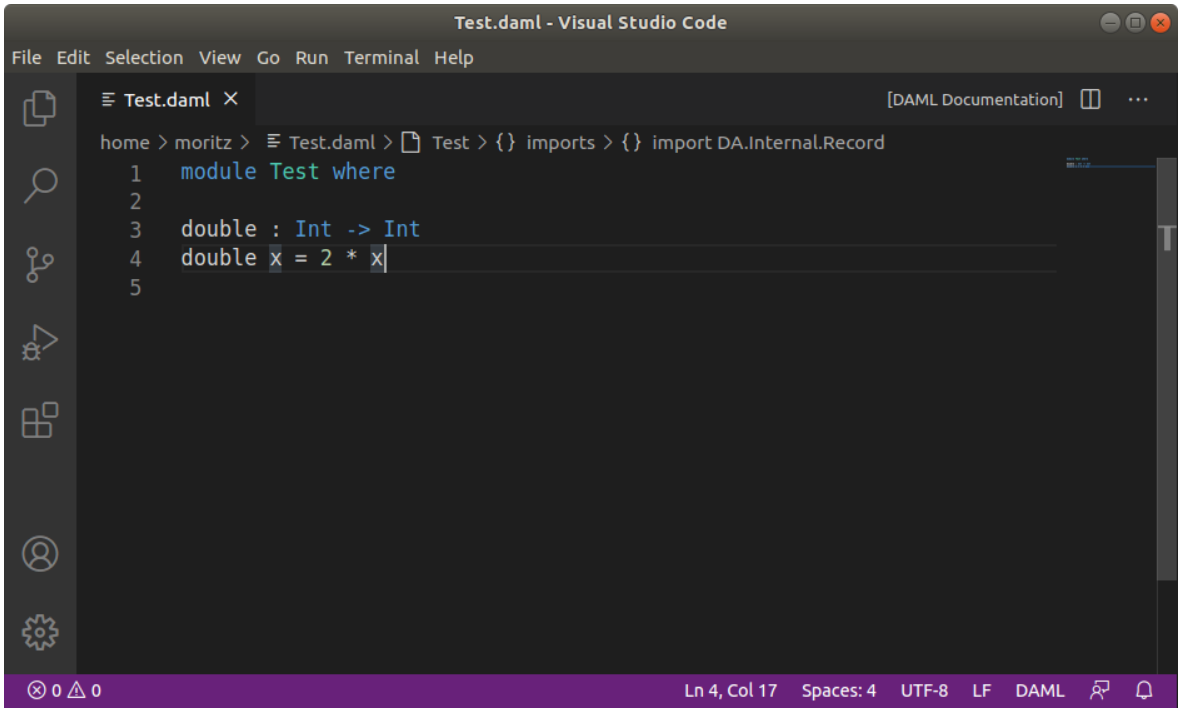


3. Open a new file (⌘N) and save it (⌘S) as `Test.daml`.

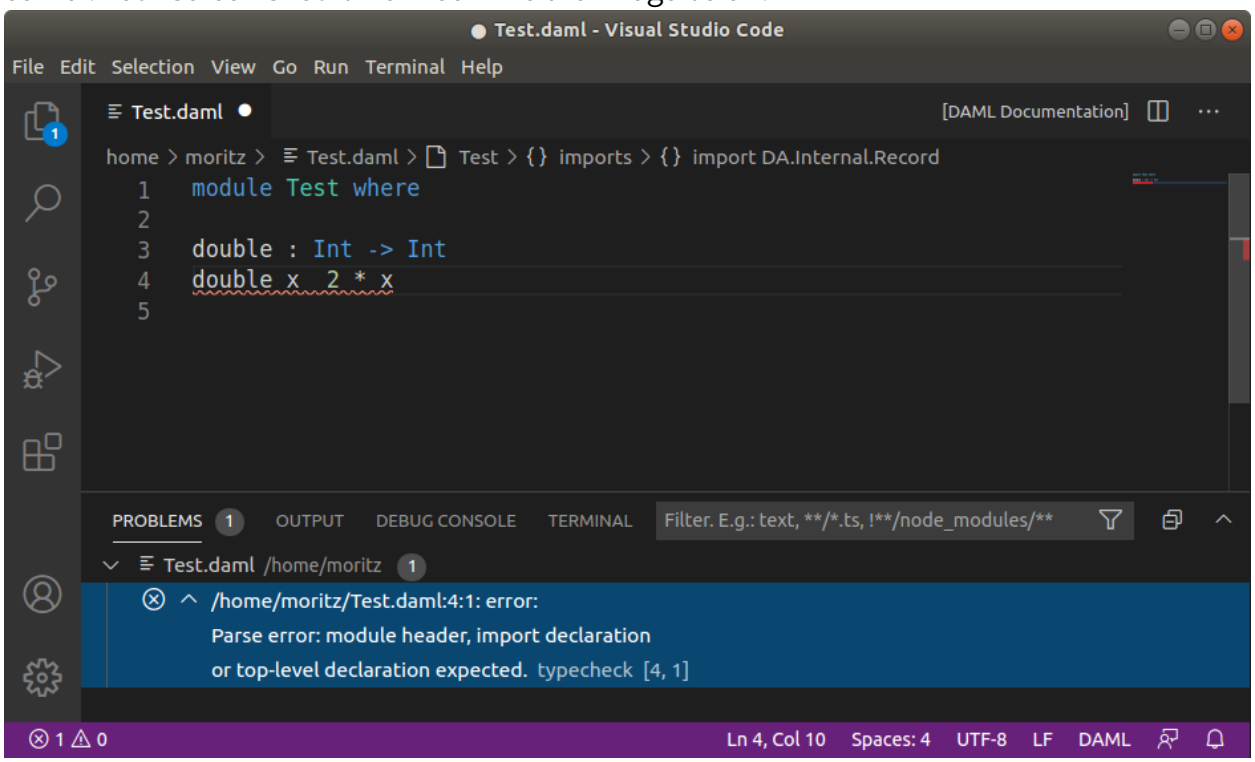
4. Copy the following code into your file:

```
module Test where

double : Int -> Int
double x = 2 * x
```

Your screen should now look like the image below.



5. Introduce a parse error by deleting the = sign and then clicking the symbol on the lower-left corner. Your screen should now look like the image below.



6. Remove the parse error by restoring the = sign.

We recommend reviewing the Visual Studio Code documentation to learn more about how to use it. To learn more about Daml, see *Language Reference*.

1.15.5.3 Supported Features

Visual Studio Code provides many helpful features for editing Daml files and we recommend reviewing Visual Studio Code Basics and Visual Studio Code Keyboard Shortcuts for OS X. The Daml Studio extension for Visual Studio Code provides the following Daml-specific features:

Symbols and Problem Reporting

Use the commands listed below to navigate between symbols, rename them, and inspect any problems detected in your Daml files. Symbols are identifiers such as template names, lambda arguments, variables, and so on.

Command	Shortcut (OS X)
Go to Definition	F12
Peek Definition	⌥F12
Rename Symbol	F2
Go to Symbol in File	⇧⌘O
Go to Symbol in Workspace	⌘T
Find all References	⇧F12
Problems Panel	⇧⌘M

Note: You can also start a command by typing its name into the command palette (press ⇧⌘P or F1). The command palette is also handy for looking up keyboard shortcuts.

Note:

Rename Symbol, Go to Symbol in File, Go to Symbol in Workspace, and Find all References work on: choices, record fields, top-level definitions, let-bound variables, lambda arguments, and modules
Go to Definition and Peek Definition work on: top-level definitions, let-bound variables, lambda arguments, and modules

---

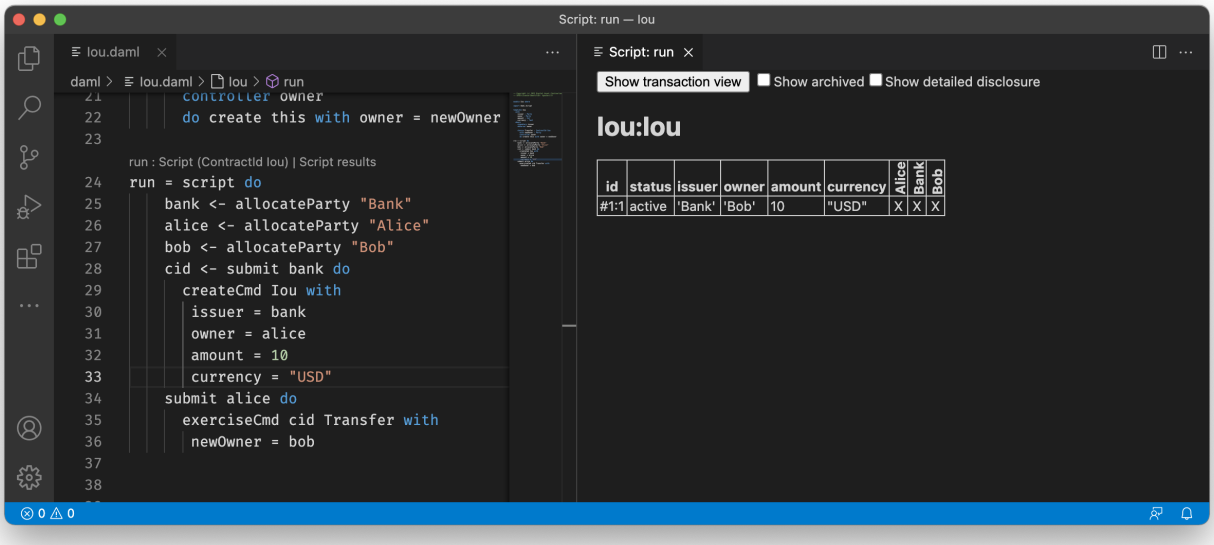
Hover Tooltips

You can hover over most symbols in the code to display additional information such as its type.

Daml Script Results

Top-level declarations of type `Script` are decorated with a `Script results` code lens. You can click on the code lens to inspect the execution transaction graph and the active contracts.

For the script from the `Iou` module, you get the following table displaying all contracts that are active at the end of the script. The first column displays the contract id. The columns afterwards represent the fields of the contract and finally you get one column per party with an `X` if the party can see the contract or a – if not.

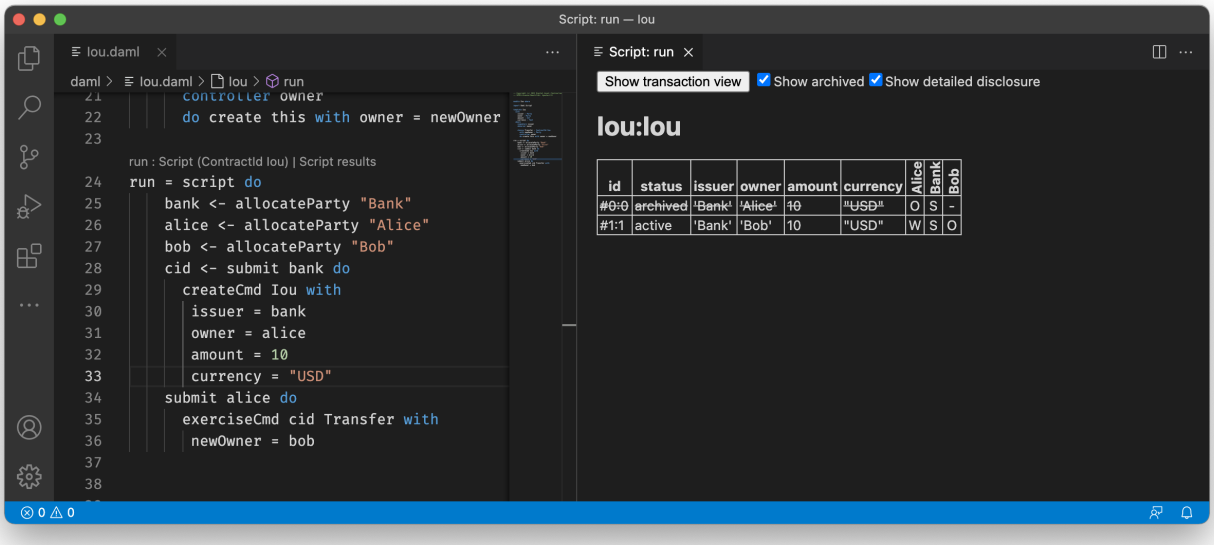


If you want more details, you can click on the *Show archived* checkbox, which extends the table to include archived contracts, and on the *Show detailed disclosure* checkbox, which displays why the contract is visible to each party, based on four categories:

1. `S`, the party sees the contract because they are a signatory on the contract.
2. `O`, the party sees the contract because they are an observer on the contract.
3. `W`, the party sees the contract because they witnessed the creation of this contract, e.g., because they are an actor on the `exercise` that created it.
4. `D`, the party sees the contract because they have been divulged the contract, e.g., because they witnessed an exercise that resulted in a `fetch` of this contract.

For details on the meaning of those four categories, refer to the *Daml Ledger Model*. For the example above, the resulting table looks as follows. You can see the archived `Bank` contract and the active `Bank` contract whose creation `Alice` has witnessed by virtue of being an actor on the `exercise` that created it.

If you want to see the detailed transaction graph you can click on the `Show transaction view` button. The transaction graph consists of transactions, each of which contain one or more updates to the ledger, that is creates and exercises. The transaction graph also records fetches of contracts.



For example a script for the `Iou` module looks as follows:

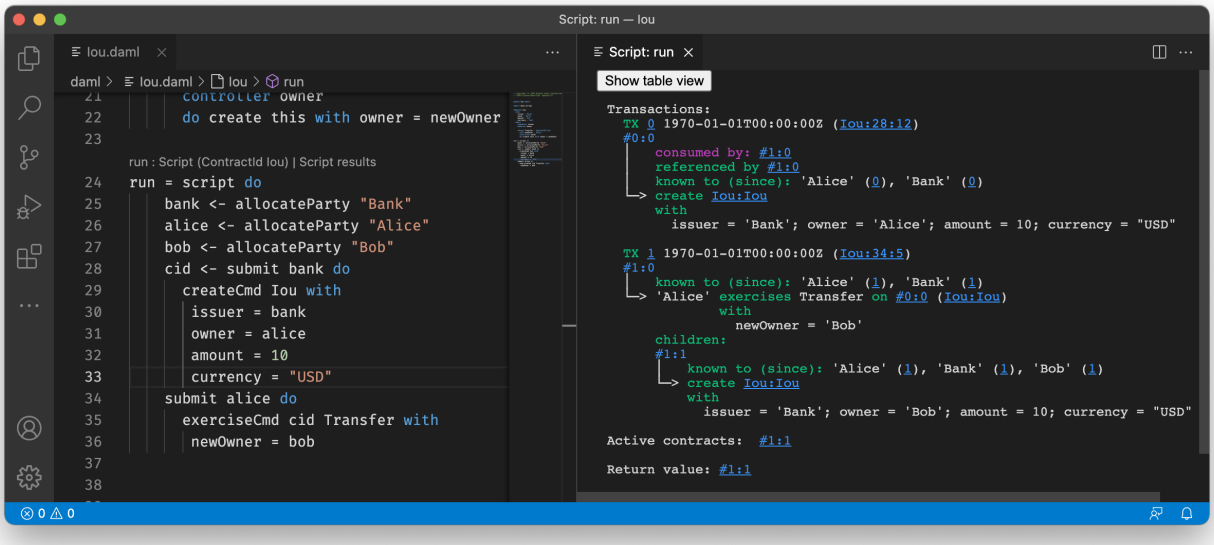


Fig. 8: Script results

Each transaction is the result of executing a step in the script. In the image below, the transaction `#0` is the result of executing the first line of the script (line 20), where the Iou is created by the bank. The following information can be gathered from the transaction:

> The result of the first script transaction `#0` was the creation of the `Iou` contract with the arguments `bank`, `10`, and `"USD"`.
> The created contract is referenced in transaction `#1`, step `0`.
> The created contract was consumed in transaction `#1`, step `0`.
> A new contract was created in transaction `#1`, step `1`, and has been divulged to parties 'Alice', 'Bob', and 'Bank'.

At the end of the script only the contract created in `#1:1` remains.
The return value from running the script is the contract identifier `#1:1`.
And finally, the contract identifiers assigned in script execution correspond to the script step that created them (e.g. `#1`).

You can navigate to the corresponding source code by clicking on the location shown in parenthesis (e.g. `Iou:25:12`, which means the `Iou` module, line 25 and column 1). You can also navigate between transactions by clicking on the transaction and contract ids (e.g. `#1:0`).

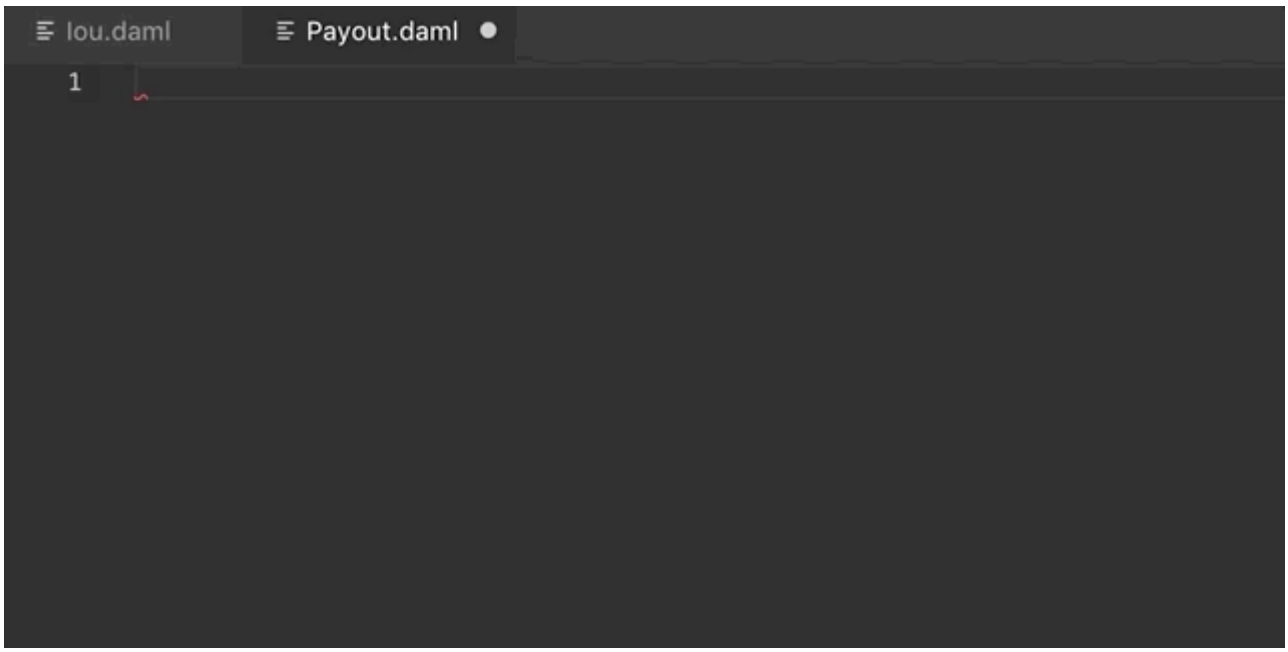
Daml Snippets

You can automatically complete a number of snippets when editing a Daml source file. By default, hitting `^-Space` after typing a Daml keyword displays available snippets that you can insert.

To define your own workflow around Daml snippets, adjust your user settings in Visual Studio Code to include the following options:

```
{
  "editor.tabCompletion": true,
  "editor.quickSuggestions": false
}
```

With those changes in place, you can simply hit `Tab` after a keyword to insert the code pattern.



You can develop your own snippets by following the instructions in Creating your own Snippets to create an appropriate `daml.json` snippet file.

1.15.5.4 Common Script Errors

During Daml execution, errors can occur due to exceptions (e.g. use of `abort`, or division by zero), or due to authorization failures. You can expect to run into the following errors when writing Daml.

When a runtime error occurs in a script execution, the script result view shows the error together with the following additional information, if available:

Location of the failed commit If the failing part of the script was a `submitCmd`, the source location of the call to `submitCmd` will be displayed.
Stack trace A list of source locations that were encountered before the error occurred. The last encountered location is the first entry in the list.
Ledger time The ledger time at which the error occurred.
Partial transaction The transaction that is being constructed, but not yet committed to the ledger.
Committed transaction Transactions that were successfully committed to the ledger prior to the error.
Trace Any messages produced by calls to `trace` and `debug`.

Abort, Assert, and Debug

The `abort`, `assert` and `debug` inbuilt functions can be used in updates and scripts. All three can be used to output messages, but `abort` and `assert` can additionally halt the execution:

```
abortTest = script do
  debug "hello, world!"
  abort "stop"
```

```
Script execution failed:
  Unhandled exception:  DA.Exception.GeneralError:GeneralError with
                          message = "stop"

Ledger time: 1970-01-01T00:00:00Z

Trace:
  "hello, world!"
```

Missing Authorization on Create

If a contract is being created without approval from all authorizing parties the commit will fail. For example:

```
template Example
  with
    party1 : Party; party2 : Party
  where
    signatory party1
    signatory party2

example = script do
  alice <- allocateParty "Alice"
  bob <- allocateParty "Bob"
  alice `submit` createCmd Example with
```

(continues on next page)

(continued from previous page)

```
    party1 = alice
    party2 = bob
```

Execution of the example script fails due to 'Bob' being a signatory in the contract, but not authorizing the create:

```
Script execution failed:
  #0: create of CreateAuthFailure:Example at unknown source
      failed due to a missing authorization from 'Bob'

Ledger time: 1970-01-01T00:00:00Z

Partial transaction:
  Sub-transactions:
     #0
     └─> Alice creates CreateAuthFailure:Example
         with
           party1 = 'Alice'; party2 = 'Bob'
```

To create the Example contract one would need to bring both parties to authorize the creation via a choice, for example 'Alice' could create a contract giving 'Bob' the choice to create the 'Example' contract.

Missing Authorization on Exercise

Similarly to creates, exercises can also fail due to missing authorizations when a party that is not a controller of a choice exercises it.

```
template Example
  with
    owner : Party
    friend : Party
  where
    signatory owner
    observer friend

    choice Consume : ()
      controller owner
      do return ()

    choice Hello : ()
      controller friend
      do return ()

example = script do
  alice <- allocateParty "Alice"
  bob <- allocateParty "Bob"
  cid <- alice `submit` createCmd Example with
    owner = alice
    friend = bob
  bob `submit` exerciseCmd cid Consume
```

The execution of the example script fails when 'Bob' tries to exercise the choice 'Consume' of which he is not a controller

```
Script execution failed:
  #1: exercise of Consume in ExerciseAuthFailure:Example at unknown source
      failed due to a missing authorization from 'Alice'

Ledger time: 1970-01-01T00:00:00Z

Partial transaction:
  Failed exercise:
    exercises Consume on #0:0 (ExerciseAuthFailure:Example)
    with
  Sub-transactions:
    0
    └─> 'Alice' exercises Consume on #0:0 (ExerciseAuthFailure:Example)
                with

Committed transactions:
  TX #0 1970-01-01T00:00:00Z (unknown source)
  #0:0
  │   disclosed to (since): 'Alice' (#0), 'Bob' (#0)
  └─> 'Alice' creates ExerciseAuthFailure:Example
              with
                owner = 'Alice'; friend = 'Bob'
```

From the error we can see that the parties authorizing the exercise ('Bob') is not a subset of the required controlling parties.

Contract Not Visible

Contract not being visible is another common error that can occur when a contract that is being fetched or exercised has not been disclosed to the committing party. For example:

```
template Example
  with owner: Party
  where
    signatory owner

    choice Consume : ()
      controller owner
      do return ()

example = script do
  alice <- allocateParty "Alice"
  bob <- allocateParty "Bob"
  cid <- alice `submit` createCmd Example with owner = alice
  bob `submit` exerciseCmd cid Consume
```

In the above script the 'Example' contract is created by 'Alice' and makes no mention of the party 'Bob' and hence does not cause the contract to be disclosed to 'Bob'. When 'Bob' tries to exercise the contract the following error would occur:

```
Script execution failed:
  Attempt to fetch or exercise a contract not visible to the reading parties.
  Contract:  #0:0 (NotVisibleFailure:Example)
  actAs: 'Bob'
```

(continues on next page)

(continued from previous page)

```
  readAs:
  Disclosed to: 'Alice'

Ledger time: 1970-01-01T00:00:00Z

Partial transaction:

Committed transactions:
  TX #0 1970-01-01T00:00:00Z (unknown source)
  #0:0
  │   disclosed to (since): 'Alice' (#0)
  └─> 'Alice' creates NotVisibleFailure:Example
              with
                owner = 'Alice'
```

To fix this issue the party 'Bob' should be made a controlling party in one of the choices.

1.15.5.5 Work with Multiple Packages

Often a Daml project consists of multiple packages, e.g., one containing your templates and one containing a Daml trigger so that you can keep the templates stable while modifying the trigger. It is possible to work on multiple packages in a single session of Daml studio but you have to keep some things in mind. You can see the directory structure of a simple multi-package project consisting of two packages `pkga` and `pkgb` below:

```
.
├── daml.yaml
├── pkga
│   ├── daml
│   │   └── A.daml
│   └── daml.yaml
└── pkgb
    ├── daml
    │   └── B.daml
    └── daml.yaml
```

`pkga` and `pkgb` are regular Daml projects with a `daml.yaml` and a Daml module. In addition to the `daml.yaml` files for the respective packages, you also need to add a `daml.yaml` to the root of your project. This file only needs to specify the SDK version. Replace `X.Y.Z` by the SDK version you specified in the `daml.yaml` files of the individual packages.

```
sdk-version: X.Y.Z
```

You can then open Daml Studio once in the root of your project and work on files in both packages. Note that if `pkgb` refers to `pkga.dar` in its `dependencies` field, changes will not be picked up automatically. This is always the case even if you open Daml Studio in `pkgb`. However, for multi-package projects there is an additional caveat: You have to both rebuild `pkga.dar` using `daml build` and then build `pkgb` using `daml build` before restarting Daml Studio.

1.15.6 Daml Sandbox

The Daml Sandbox, or Sandbox for short, is a simple ledger implementation that enables rapid application prototyping by simulating a Daml Ledger.

You can start Sandbox together with *Navigator* using the `daml start` command in a Daml project. This command will compile the Daml file and its dependencies as specified in the `daml.yaml`. It will then launch Sandbox passing the just obtained DAR packages. The script specified in the `init-script` field in `daml.yaml` will be loaded into the ledger. Finally, it launches the navigator connecting it to the running Sandbox.

It is possible to execute the Sandbox launching step in isolation by typing `daml sandbox`.

Sandbox can also be run manually as in this example:

```
$ daml sandbox --dar Main.dar --static-time
Starting Canton sandbox.
Listening at port 6865
Uploading .daml/dist/foobar-0.0.1.dar to localhost:6865
DAR upload succeeded.
Canton sandbox is ready.
```

Behind the scenes, Sandbox spins up a Canton ledger with an in-memory participant `sandbox` and an in-memory domain `local`. You can pass additional Canton configuration files via `-c`. This option can be specified multiple times and the resulting configuration files will be merged.

```
$ daml sandbox -c path/to/canton/config
```

1.15.6.1 Run With Authorization

By default, Sandbox accepts all valid ledger API requests without performing any request authorization.

To start Sandbox with authorization using JWT-based access tokens as described in the *Authorization documentation*, create a config file that specifies the type of authorization service and the path to the certificate.

Listing 26: auth.conf

```
canton.participants.sandbox.ledger-api.auth-services = [{
    // type can be
    //   jwt-rs-256-crt
    //   jwt-es-256-crt
    //   jwt-es-512-crt
    //   jwt-rs-256-jwks with an additional url
    //   unsafe-jwt-hmac-256 with an additional secret
    type = jwt-rs-256-crt
    certificate = my-certificate.cert
}]
```

The settings under `auth-services` are described in detail in API configuration documentation

Generate JSON Web Tokens (JWT)

To generate access tokens for testing purposes, use the jwt.io web site.

Generate RSA keys

To generate RSA keys for testing purposes, use the following command

```
openssl req -nodes -new -x509 -keyout sandbox.key -out sandbox.crt
```

which generates the following files:

`sandbox.key`: the private key in PEM/DER/PKCS#1 format
`sandbox.crt`: a self-signed certificate containing the public key, in PEM/DER/X.509 Certificate format

Generate EC keys

To generate keys to be used with ES256 for testing purposes, use the following command

```
openssl req -x509 -nodes -days 3650 -newkey ec:<(openssl ecparam -name
↪prime256v1) -keyout ecdsa256.key -out ecdsa256.crt
```

which generates the following files:

`ecdsa256.key`: the private key in PEM/DER/PKCS#1 format
`ecdsa256.crt`: a self-signed certificate containing the public key, in PEM/DER/X.509 Certificate format

Similarly, you can use the following command for ES512 keys:

```
openssl req -x509 -nodes -days 3650 -newkey ec:<(openssl ecparam -name secp521r1)
↪-keyout ecdsa512.key -out ecdsa512.crt
```

1.15.6.2 Run With TLS

To enable TLS, you need to specify the private key for your server and the certificate chain. This enables TLS for both the Ledger API and the Canton Admin API. When enabling client authentication, you also need to specify client certificates which can be used by Canton's internal processes. Note that the identity of the application will not be proven by using this method, i.e. the *application_id* field in the request is not necessarily correlated with the CN (Common Name) in the certificate. Below, you can see an example config. For more details on TLS, refer to *Canton's documentation on TLS*.

Listing 27: tls.conf

```
canton.participants.sandbox.ledger-api {
  tls {
    // the certificate to be used by the server
    cert-chain-file = "./tls/participant.crt"
    // private key of the server
    private-key-file = "./tls/participant.pem"
```

(continues on next page)

(continued from previous page)

```
    // trust collection, which means that all client certificates will be
↪verified using the trusted
    // certificates in this store. if omitted, the JVM default trust store is
↪used.
    trust-collection-file = "./tls/root-ca.crt"
    // define whether clients need to authenticate as well (default not)
    client-auth = {
      // none, optional and require are supported
      type = require
      // If clients are required to authenticate as well, we need to provide a
↪client
      // certificate and the key, as Canton has internal processes that need to
↪connect to these
      // APIs. If the server certificate is trusted by the trust-collection, then
↪you can
      // just use the server certificates. Otherwise, you need to create separate
↪ones.
      admin-client {
        cert-chain-file = "./tls/admin-client.crt"
        private-key-file = "./tls/admin-client.pem"
      }
    }
  }
}
```

1.15.6.3 Command-line Reference

To start Sandbox, run: `daml sandbox [options] [-c canton.config]`.

To see all the available options, run `daml sandbox --help`. Note that this will show you the options of the Sandbox wrapper around Canton. To see options of the underlying Canton runner, use `daml sandbox --canton-help`.

1.15.6.4 Metrics

Enable and Configure Reporting

You can enable metrics reporting via Prometheus using the following configuration file.

Listing 28: metrics.conf

```
canton.monitoring.metrics.reporters = [{
  type = prometheus
  address = "localhost" // default
  port = 9000 // default
}]
```

For other options and more details refer to the *Canton documentation*.

Types of Metrics

This is a list of type of metrics with all data points recorded for each. Use this as a reference when reading the list of metrics.

Gauge

An individual instantaneous measurement.

Counter

Number of occurrences of some event.

Meter

A meter tracks the number of times a given event occurred. The following data points are kept and reported by any meter.

`<metric.qualified.name>.count`: number of registered data points overall
`<metric.qualified.name>.m1_rate`: number of registered data points per minute
`<metric.qualified.name>.m5_rate`: number of registered data points every 5 minutes
`<metric.qualified.name>.m15_rate`: number of registered data points every 15 minutes
`<metric.qualified.name>.mean_rate`: mean number of registered data points

Histogram

An histogram records aggregated statistics about collections of events. The exact meaning of the number depends on the metric (e.g. timers are histograms about the time necessary to complete an operation).

`<metric.qualified.name>.mean`: arithmetic mean
`<metric.qualified.name>.stddev`: standard deviation
`<metric.qualified.name>.p50`: median
`<metric.qualified.name>.p75`: 75th percentile
`<metric.qualified.name>.p95`: 95th percentile
`<metric.qualified.name>.p98`: 98th percentile
`<metric.qualified.name>.p99`: 99th percentile

`<metric.qualified.name>.p999`: 99.9th percentile
`<metric.qualified.name>.min`: lowest registered value overall
`<metric.qualified.name>.max`: highest registered value overall

Histograms only keep a small *reservoir* of statistically relevant data points to ensure that metrics collection can be reasonably accurate without being too taxing resource-wise.

Unless mentioned otherwise all histograms (including timers, mentioned below) use exponentially decaying reservoirs (i.e. the data is roughly relevant for the last five minutes of recording) to ensure that recent and possibly operationally relevant changes are visible through the metrics reporter.

Note that `min` and `max` values are not affected by the reservoir sampling policy.

You can read more about reservoir sampling and possible associated policies in the Dropwizard Metrics library documentation.

Timers

A timer records all metrics registered by a meter and by an histogram, where the histogram records the time necessary to execute a given operation (unless otherwise specified, the precision is nanoseconds and the unit of measurement is milliseconds).

Database Metrics

A database metric is a collection of simpler metrics that keep track of relevant numbers when interacting with a persistent relational store.

These metrics are:

`<metric.qualified.name>.wait` (timer): time to acquire a connection to the database
`<metric.qualified.name>.exec` (timer): time to run the query and read the result
`<metric.qualified.name>.query` (timer): time to run the query
`<metric.qualified.name>.commit` (timer): time to perform the commit
`<metric.qualified.name>.translation` (timer): if relevant, time necessary to turn serialized Daml-LF values into in-memory objects

List of Metrics

The following is a non-exhaustive list of selected metrics that can be particularly important to track. Note that not all the following metrics are available unless you run the sandbox with a PostgreSQL backend.

`daml.commands.delayed_submissions`

A meter. Number of delayed submissions (submission that have been evaluated to transaction with a ledger time farther in the future than the expected latency).

`daml.commands.failed_command_interpretations`

A meter. Number of commands that have been deemed unacceptable by the interpreter and thus rejected (e.g. double spends)

`daml.commands.submissions`

A timer. Time to fully process a submission (validation, deduplication and interpretation) before it's handed over to the ledger to be finalized (either committed or rejected).

`daml.commands.valid_submissions`

A meter. Number of submission that pass validation and are further sent to deduplication and interpretation.

`daml.commands.validation`

A timer. Time to validate submitted commands before they are fed to the Daml interpreter.

`daml.commands.input_buffer_capacity`

A counter. The capacity of the queue accepting submissions on the CommandService.

`daml.commands.input_buffer_length`

A counter. The number of currently pending submissions on the CommandService.

`daml.commands.input_buffer_delay`

A timer. Measures the queuing delay for pending submissions on the CommandService.

`daml.commands.max_in_flight_capacity`

A counter. The capacity of the queue tracking completions on the CommandService.

`daml.commands.max_in_flight_length`

A counter. The number of currently pending completions on the CommandService.

`daml.execution.get_lf_package`

A timer. Time spent by the engine fetching the packages of compiled Daml code necessary for interpretation.

`daml.execution.lookup_active_contract_count_per_execution`

A histogram. Number of active contracts fetched for each processed transaction.

`daml.execution.lookup_active_contract_per_execution`

A timer. Time to fetch all active contracts necessary to process each transaction.

`daml.execution.lookup_active_contract`

A timer. Time to fetch each individual active contract during interpretation.

`daml.execution.lookup_contract_key_count_per_execution`

A histogram. Number of contract keys looked up for each processed transaction.

`daml.execution.lookup_contract_key_per_execution`

A timer. Time to lookup all contract keys necessary to process each transaction.

`daml.execution.lookup_contract_key`

A timer. Time to lookup each individual contract key during interpretation.

daml.execution.retry

A meter. Overall number of interpretation retries attempted due to mismatching ledger effective time.

daml.execution.total

A timer. Time spent interpreting a valid command into a transaction ready to be submitted to the ledger for finalization.

daml.index.db.connection.api.server.pool

This namespace holds a number of interesting metrics about the connection pool used to communicate with the persistent store that underlies the index.

These metrics include:

- `daml.index.db.connection.api.server.pool.Wait` (timer): time spent waiting to acquire a connection
- `daml.index.db.connection.api.server.pool.Usage` (histogram): time spent using each acquired connection
- `daml.index.db.connection.api.server.pool.TotalConnections` (gauge): number or total connections
- `daml.index.db.connection.api.server.pool.IdleConnections` (gauge): number of idle connections
- `daml.index.db.connection.api.server.pool.ActiveConnections` (gauge): number of active connections
- `daml.index.db.connection.api.server.pool.PendingConnections` (gauge): number of threads waiting for a connection

daml.index.db.get_active_contracts

A database metric. Time spent retrieving a page of active contracts to be served from the active contract service. The page size is configurable, please look at the CLI reference.

daml.index.db.get_completions

A database metric. Time spent retrieving a page of command completions to be served from the command completion service. The page size is configurable, please look at the CLI reference.

`daml.index.db.get_flat_transactions`

A database metric. Time spent retrieving a page of flat transactions to be streamed from the transaction service. The page size is configurable, please look at the CLI reference.

`daml.index.db.get_ledger_end`

A database metric. Time spent retrieving the current ledger end. The count for this metric is expected to be very high and always increasing as the indexed is queried for the latest updates.

`daml.index.db.get_ledger_id`

A database metric. Time spent retrieving the ledger identifier.

`daml.index.db.get_transaction_trees`

A database metric. Time spent retrieving a page of flat transactions to be streamed from the transaction service. The page size is configurable, please look at the CLI reference.

`daml.index.db.load_all_parties`

A database metric. Load the currently allocated parties so that they are served via the party management service.

`daml.index.db.load_archive`

A database metric. Time spent loading a package of compiled Daml code so that it's given to the Daml interpreter when needed.

`daml.index.db.load_configuration_entries`

A database metric. Time to load the current entries in the log of configuration entries. Used to verify whether a configuration has been ultimately set.

`daml.index.db.load_package_entries`

A database metric. Time to load the current entries in the log of package uploads. Used to verify whether a package has been ultimately uploaded.

daml.index.db.load_packages

A database metric. Load the currently uploaded packages so that they are served via the package management service.

daml.index.db.load_parties

A database metric. Load the currently allocated parties so that they are served via the party service.

daml.index.db.load_party_entries

A database metric. Time to load the current entries in the log of party allocations. Used to verify whether a party has been ultimately allocated.

daml.index.db.lookup_active_contract

A database metric. Time to fetch one contract on the index to be used by the Daml interpreter to evaluate a command into a transaction.

daml.index.db.lookup_configuration

A database metric. Time to fetch the configuration so that it's served via the configuration management service.

daml.index.db.lookup_contract_by_key

A database metric. Time to lookup one contract key on the index to be used by the Daml interpreter to evaluate a command into a transaction.

daml.index.db.lookup_flat_transaction_by_id

A database metric. Time to lookup a single flat transaction by identifier to be served by the transaction service.

daml.index.db.lookup_maximum_ledger_time

A database metric. Time spent looking up the ledger effective time of a transaction as the maximum ledger time of all active contracts involved to ensure causal monotonicity.

`daml.index.db.lookup_transaction_tree_by_id`

A database metric. Time to lookup a single transaction tree by identifier to be served by the transaction service.

`daml.index.db.store_configuration_entry`

A database metric. Time spent persisting a change in the ledger configuration provided through the configuration management service.

`daml.index.db.store_ledger_entry`

A database metric. Time spent persisting a transaction that has been successfully interpreted and is final.

`daml.index.db.store_package_entry`

A database metric. Time spent storing a Daml package uploaded through the package management service.

`daml.index.db.store_party_entry`

A database metric. Time spent storing party information as part of the party allocation endpoint provided by the party management service.

`daml.index.db.store_rejection`

A database metric. Time spent persisting the information that a given command has been rejected.

`daml.indexer.last_received_record_time`

A monotonically increasing integer value that represents the record time of the last event ingested by the index db. It is measured in milliseconds since the EPOCH time.

`daml.indexer.last_received_offset`

A string value representing the last ledger offset ingested by the index db. It is only available on metrics backends that support strings. In particular it is not available in Prometheus.

daml.indexer.current_record_time_lag

A lag between the record time of a transaction and the wall-clock time registered at the ingestion time to the index db. Depending on the systemic clock skew between different machines, this value can be negative.

daml.indexer.ledger_end_sequential_id

A monotonically increasing integer value representing the sequential id ascribed to the most recent ledger event ingested by the index db. Please note, that only a subset of all ledger events are ingested and given a sequential id. These are: creates, consuming exercises, non-consuming exercises and divulgence events. This value can be treated as a counter of all such events visible to a given participant.

daml.lapi

Every metrics under this namespace is a timer, one for each service exposed by the Ledger API, in the format:

`daml.lapi.service_name.service_endpoint`

As in the following example:

`daml.lapi.command_service.submit_and_wait`

Single call services return the time to serve the request, streaming services measure the time to return the first response.

daml.lapi.return_status

This namespace contains counters for various gRPC return status codes in the following format

`daml.lapi.return_status.<gRPC status code>`

As in the following example:

`daml.lapi.return_status.ABORTED`

daml.services

Every metrics under this namespace is a timer, one for each endpoint exposed by the index, read or write service. Metrics are in the format:

`daml.services.service_name.service_endpoint`

The following example demonstrates a metric for transactions submitted over the write service:

`daml.services.write.submit_transaction`

Single call services return the time to serve the request, streaming services measure the time to return the first response.

`jvm`

Under the `jvm` namespace there is a collection of metrics that tracks important measurements about the JVM that the sandbox is running on, including CPU usage, memory consumption and the current state of threads.

1.15.7 Navigator

The Navigator is a front-end that you can use to connect to any Daml Ledger and inspect and modify the ledger. You can use it during Daml development to explore the flow and implications of the Daml models.

The first sections of this guide cover use of the Navigator with the SDK. Refer to *Advanced Usage* for information on using Navigator outside the context of the SDK.

1.15.7.1 Navigator Functionality

Connect the Navigator to any Daml Ledger and use it to:

View templates
View active and archived contracts
Exercise choices on contracts
Advance time (This option applies only when using Navigator with the Daml Sandbox ledger.)

1.15.7.2 Starting Navigator

Navigator is included in the SDK. To launch it:

1. Start Navigator via a terminal window running *Daml Assistant* by typing `daml start`
2. The Navigator web-app is automatically started in your browser. If it fails to start, open a browser window and point it to the Navigator URL

   When running `daml start` you will see the Navigator URL. By default it will be http://localhost:7500/.

Note: Navigator is compatible with these browsers: Safari, Chrome, or Firefox.

1.15.7.3 Logging In

By default, Navigator shows a drop-down list with the users that have been created via the *user management service*. During development, it is common to create these users in a *Daml script*: that you specify in the `init-script` section of your `daml.yaml` file so it is executed on `daml start`. Most of the templates shipped with the Daml SDK already include such a setup script. Only users that have a primary party set will be displayed.

After logging in, you will interact with the ledger as the primary party of that user, meaning that you can see contracts visible to that party and submit commands (e.g. create a contract) as that party.

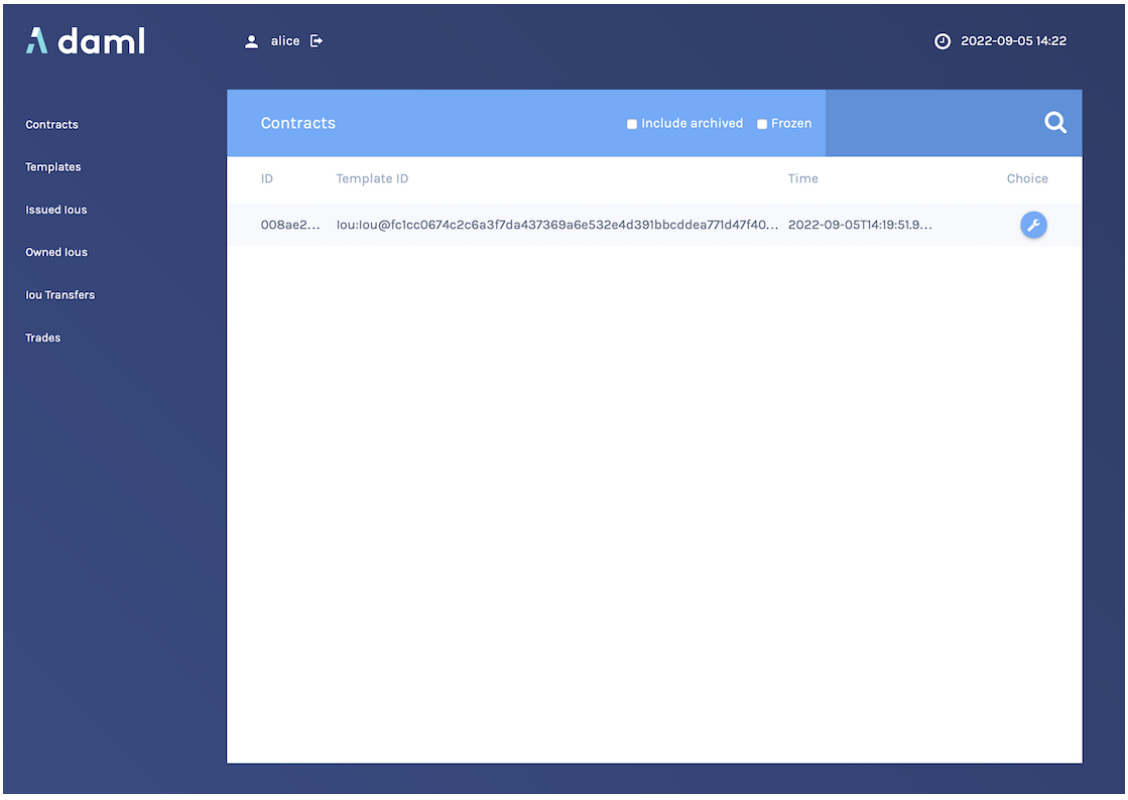
The party you are logged in as is not displayed directly. However, Navigator provides autocompletion based on the party id which starts with the party id hint so a good option is to set the party id hint

to the user id when you allocate the party in your setup script. You can see an example of that in the `skeleton` template:

```
alice <- allocatePartyWithHint "Alice" (PartyIdHint "Alice")
bob <- allocatePartyWithHint "Bob" (PartyIdHint "Bob")
aliceId <- validateUserId "alice"
bobId <- validateUserId "bob"
createUser (User aliceId (Some alice)) [CanActAs alice]
createUser (User bobId (Some bob)) [CanActAs bob]
```

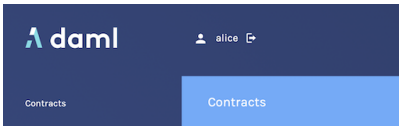
The first step in using Navigator is to use the dropdown list on the Navigator home screen to select from the available users.

The main Navigator screen will be displayed, with contracts that the primary party of this user is entitled to view in the main pane and the option to switch from contracts to templates in the pane at the left. Other options allow you to filter the display, include or exclude archived contracts, and exercise choices as described below.



To change the active user:

1. Click the name of the current user in the top left corner of the screen.
2. On the home screen, select a different user.



You can act as different users in different browser windows. Use Chrome's profile feature https://support.google.com/chrome/answer/2364824 and sign in as a different user for each Chrome profile.

Logging in as a Party

Instead of logging in by specifying a user, you can also log in by specifying a party directly. This is useful if you do not want to or cannot (because your ledger does not support user management) create users.

To do so, you can start Navigator with a flag to disable support for user management:

```
daml navigator --feature-user-management=false
```

To use this via `daml start`, you can specify it in your `daml.yaml` file:

```
navigator-options:
  - --feature-user-management=false
```

Instead of displaying a list of users on login, Navigator will display a list of parties where each party is identified by its display name.

Alternatively you can specify a fixed list of parties in your `daml.yaml` file. This will automatically disable user management and display those parties on log in. Note that you still need to allocate those parties before you can log in as them.

```
parties:
  - Alice::12201d00faa0968d7ab81e63ad6ad4ee0d31b08a3581b1d8596e68a1356f27519ccb
  - Bob::12201d00faa0968d7ab81e63ad6ad4ee0d31b08a3581b1d8596e68a1356f27519ccb
```

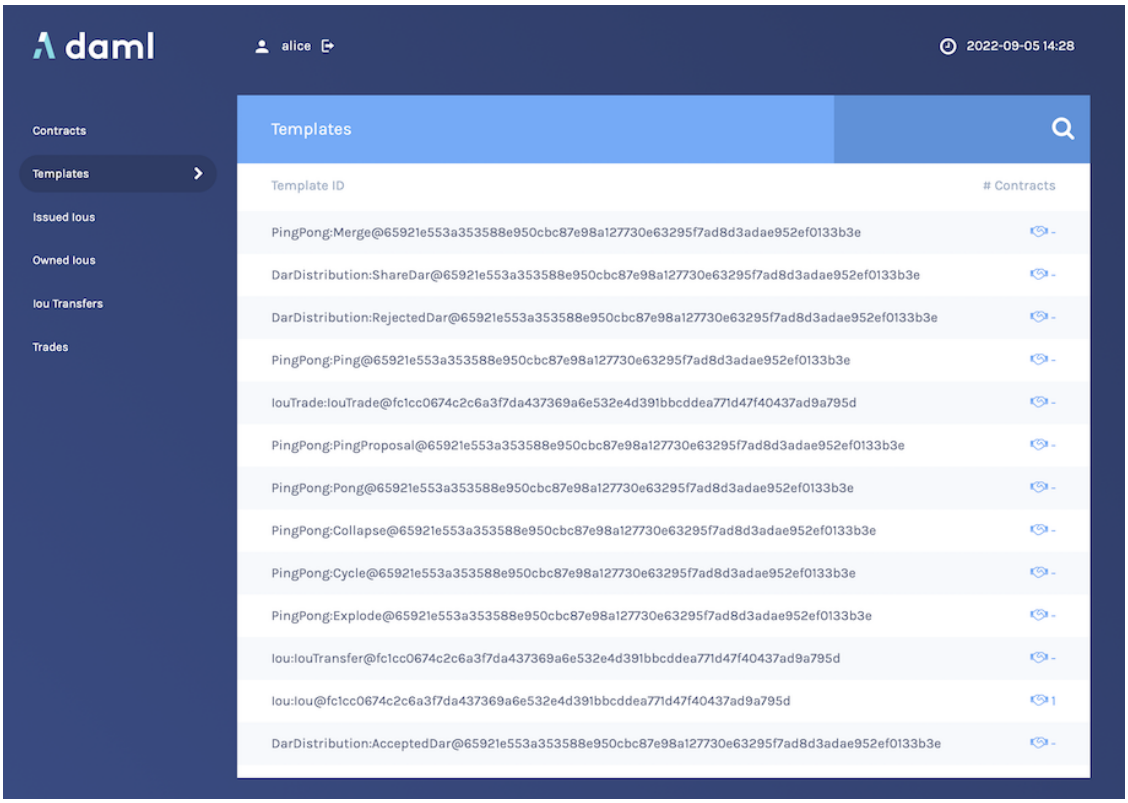
1.15.7.4 Viewing Templates or Contracts

Daml *contract templates* are models that contain the agreement statement, all the applicable parameters, and the choices that can be made in acting on that data. They specify acceptable input and the resulting output. A contract template contains placeholders rather than actual names, amounts, dates, and so on. In a *contract*, the placeholders have been replaced with actual data.

The Navigator allows you to list templates or contracts, view contracts based on a template, and view template and contract details.

Listing templates

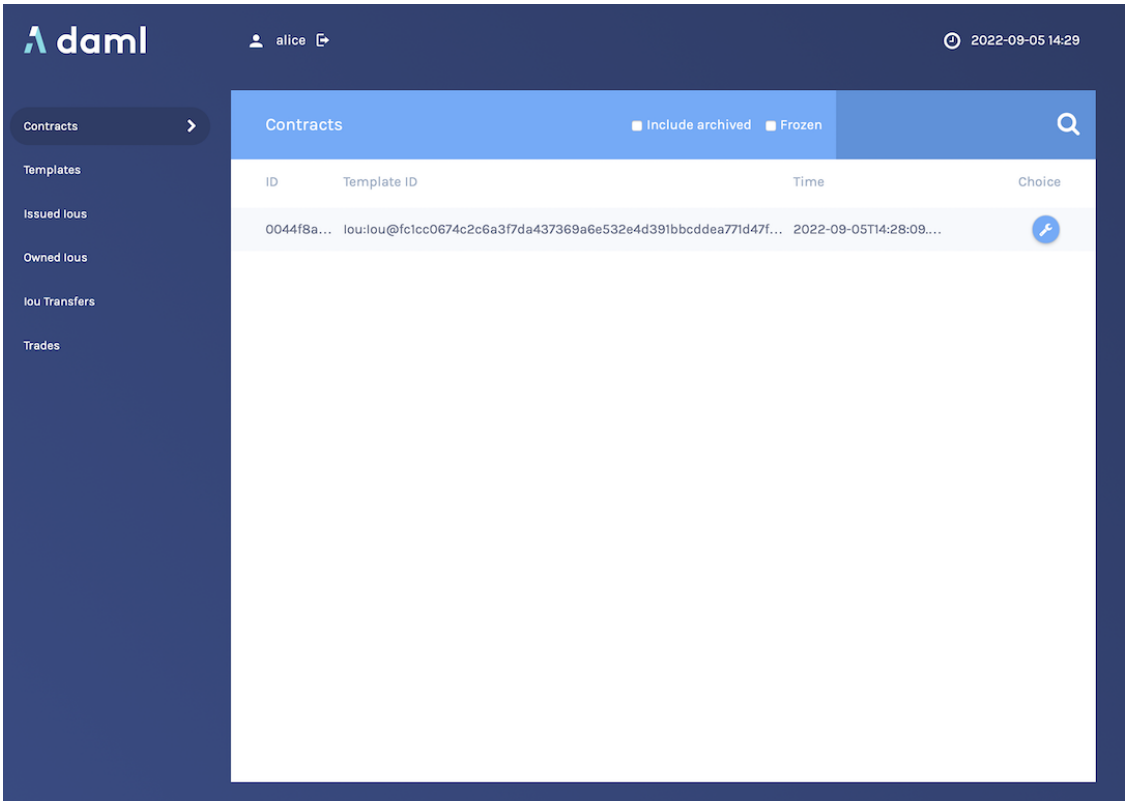
To see what contract templates are available on the ledger you are connected to, choose **Templates** in the left pane of the main Navigator screen.



Use the **Filter** field at the top right to select template IDs that include the text you enter.

Listing contracts

To view a list of available contracts, choose **Contracts** in the left pane.



In the Contracts list:

Changes to the ledger are automatically reflected in the list of contracts. To avoid the automatic updates, select the **Frozen** checkbox. Contracts will still be marked as archived, but the contracts list will not change.

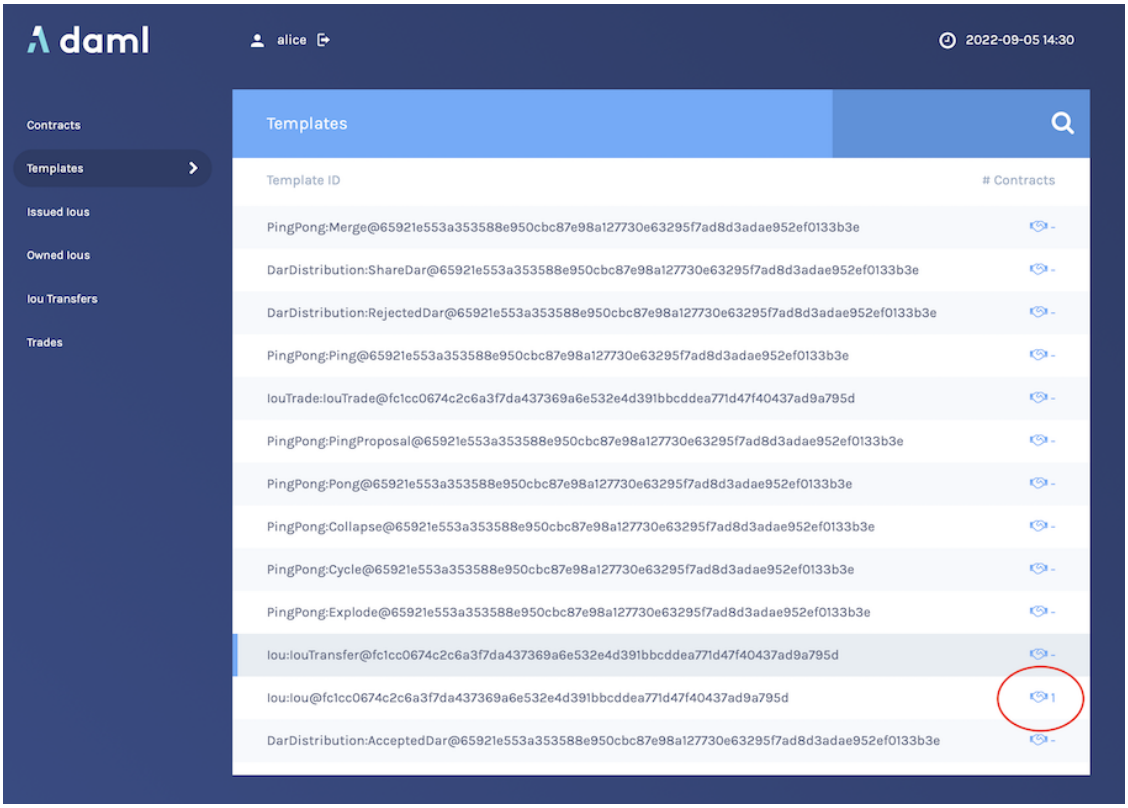
Filter the displayed contracts by entering text in the **Filter** field at the top right.
Use the **Include Archived** checkbox at the top to include or exclude archived contracts.

Viewing contracts based on a template

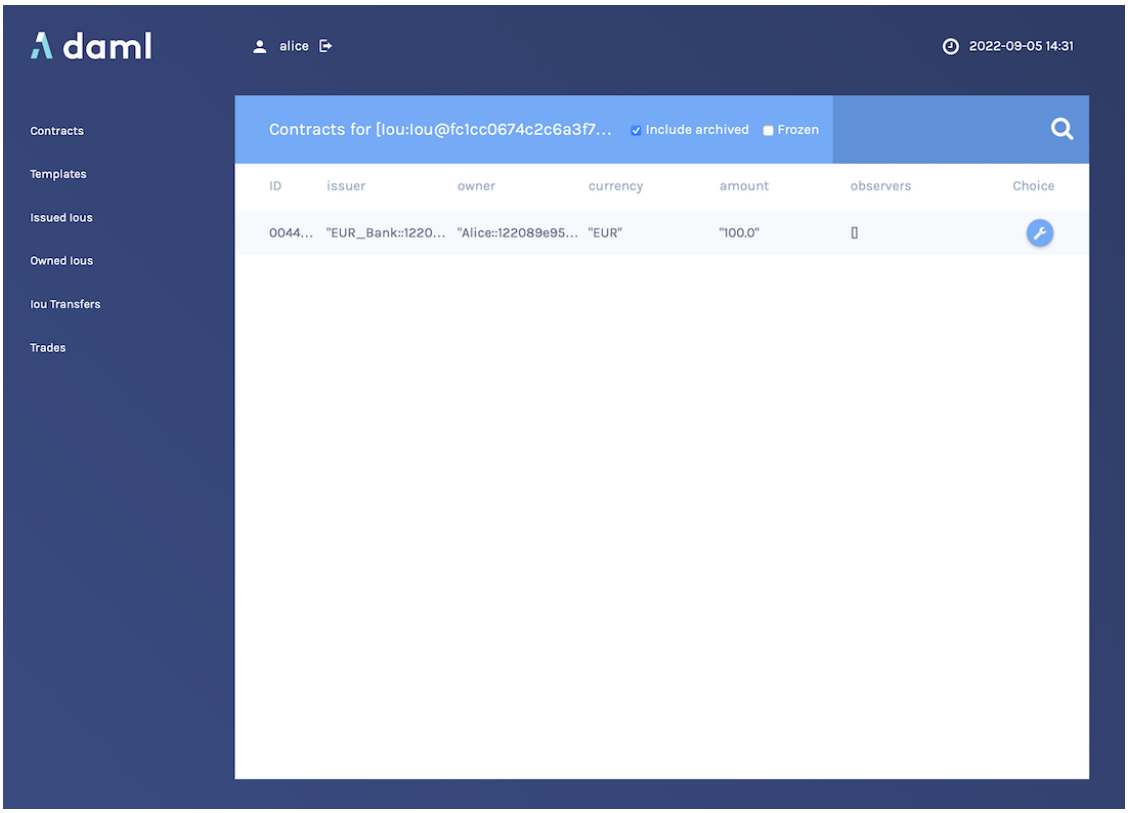
You can also view the list of contracts that are based on a particular template.

1. You will see icons to the right of template IDs in the template list with a number indicating how many contracts are based on this template.
2. Click the number to display a list of contracts based on that template.

Number of Contracts



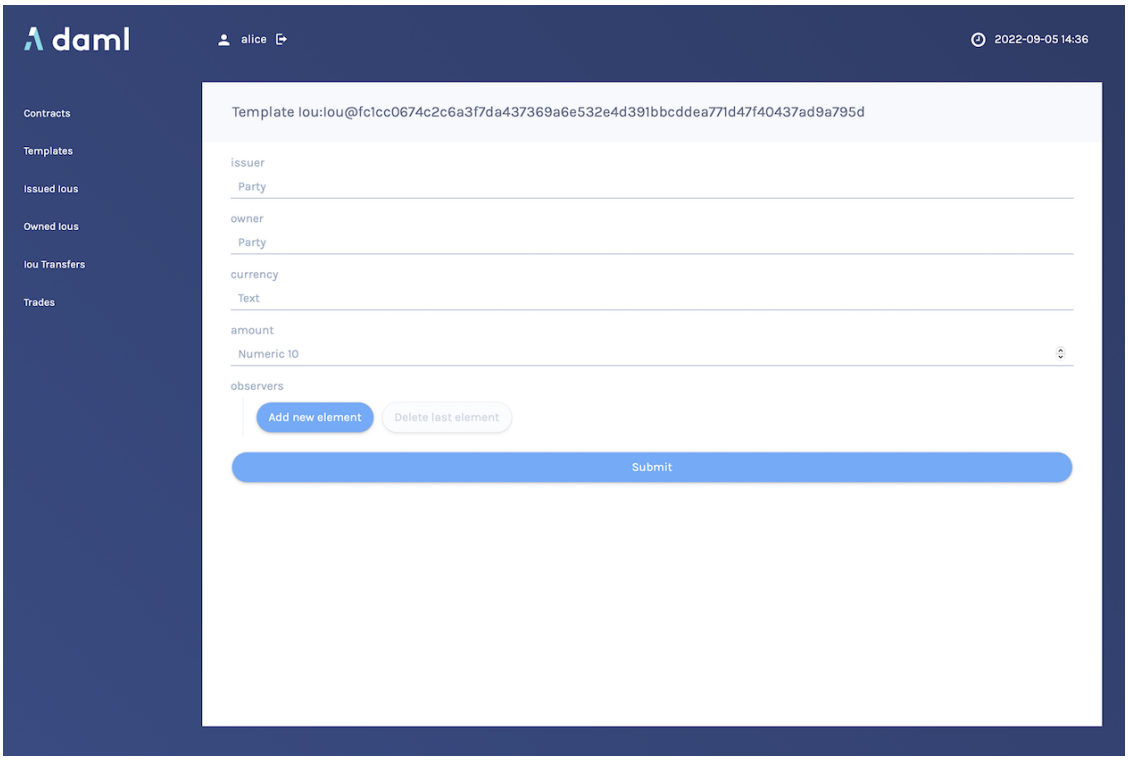
List of Contracts



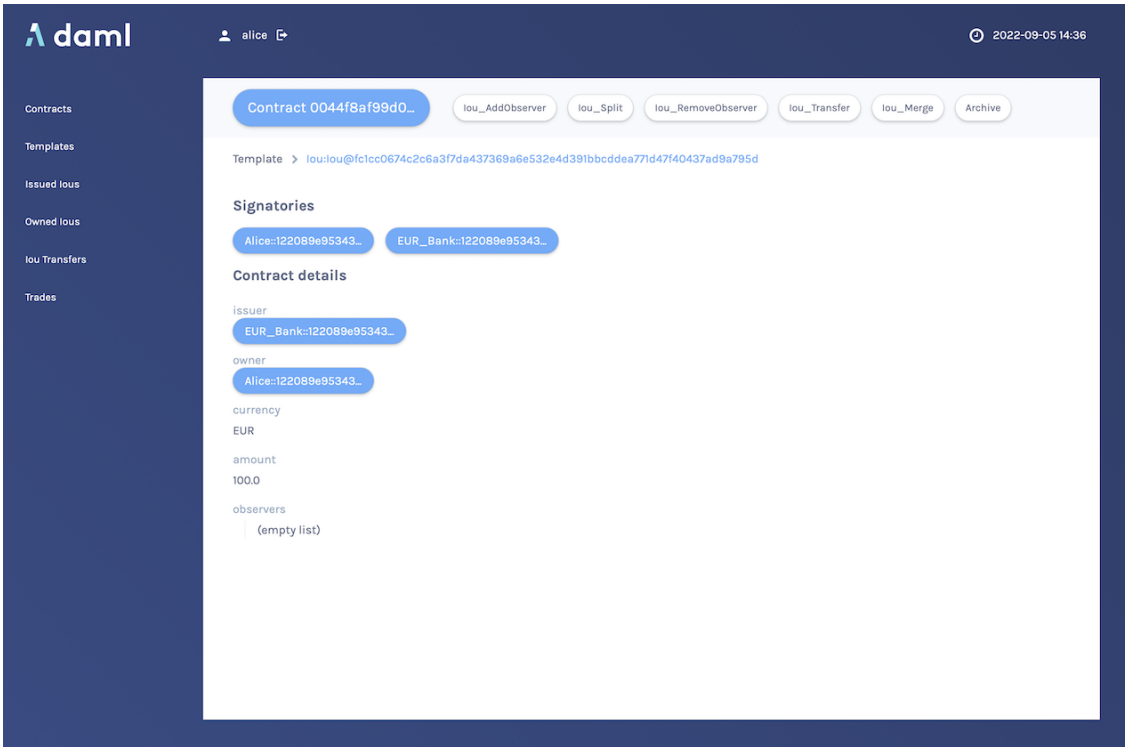
Viewing template and contract details

To view template or contract details, click on a template or contract in the list. The template or contracts detail page is displayed.

Template Details



Contract Details



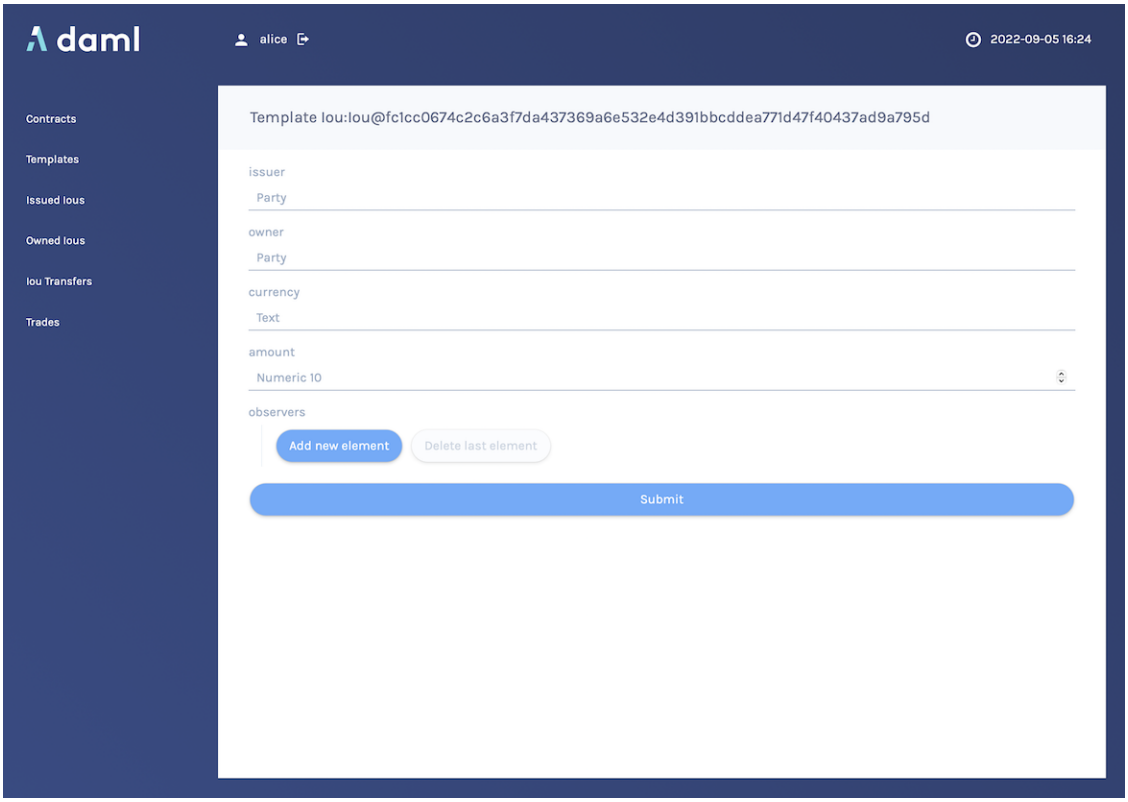
1.15.7.5 Using Navigator

Creating contracts

Contracts in a ledger are created automatically when you exercise choices. In some cases, you create a contract directly from a template. This feature can be particularly useful for testing and experimenting during development.

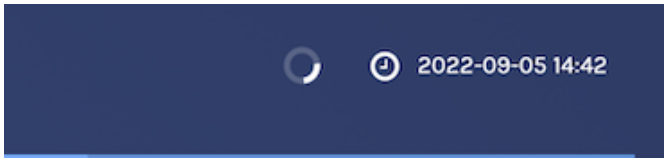
To create a contract based on a template:

1. Navigate to the template detail page as described above.
2. Complete the values in the form
3. Choose the **Submit** button.

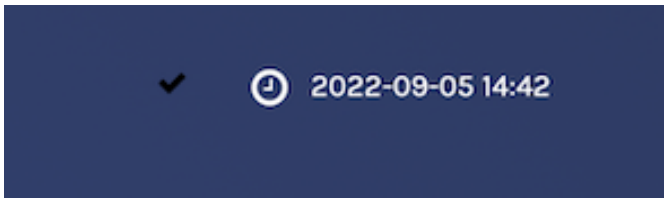


When the command has been committed to the ledger, the loading indicator in the navbar at the top will display a tick mark.

While loading



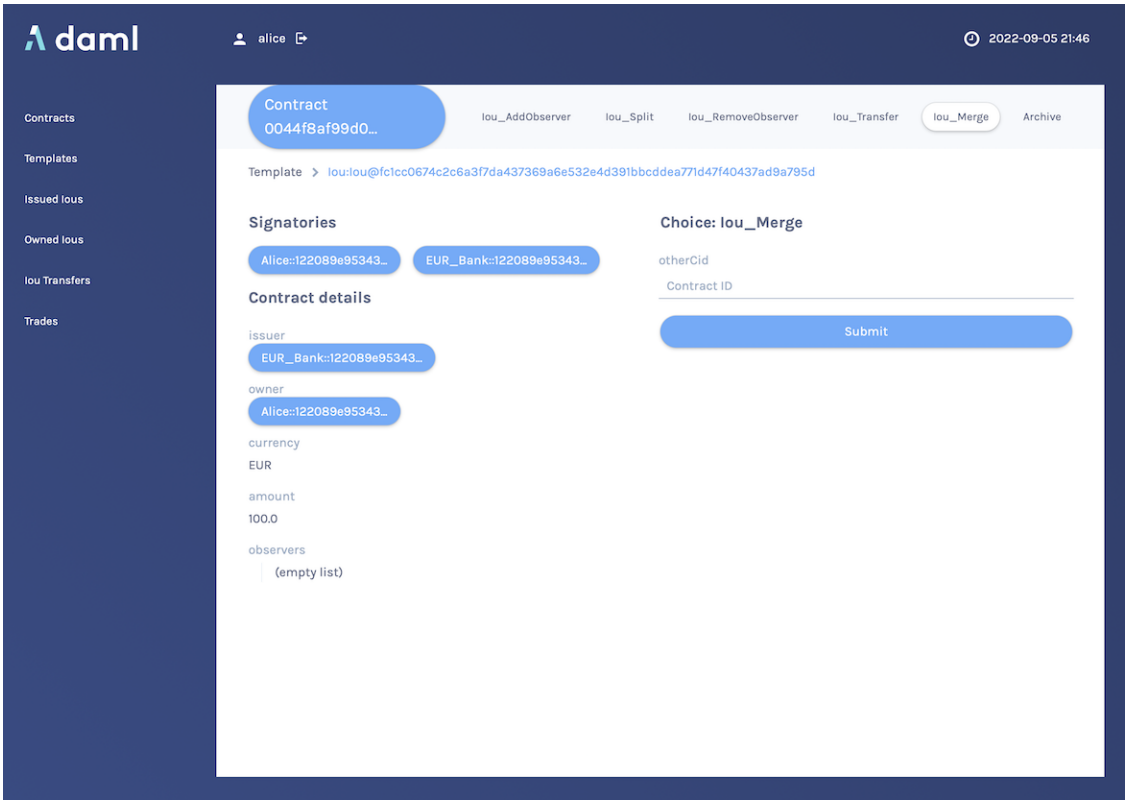
When committed to the ledger



Exercising choices

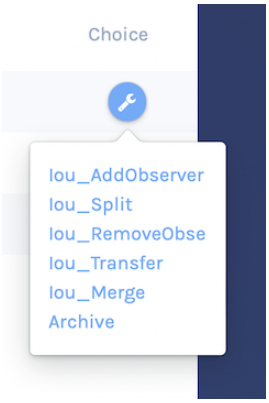
To exercise a choice:

1. Navigate to the contract details page (see above).
2. Click the choice you want to exercise in the choice list.
3. Complete the form.
4. Choose the **Submit** button.



Or

1. Navigate to the choice form by clicking the wrench icon in a contract list.
2. Select a choice.



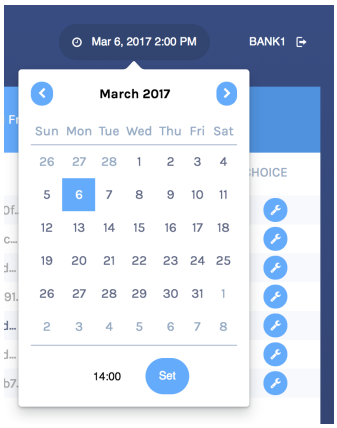
You will see the loading and confirmation indicators, as pictured above in Creating Contracts.

Advancing time

It is possible to advance time against the Daml Sandbox. (This is not true of all Daml Ledgers.) This advance-time functionality can be useful when testing, for example, when entering a trade on one date and settling it on a later date.

To advance time:

1. Click on the ledger time indicator in the navbar at the top of the screen.
2. Select a new date / time.
3. Choose the **Set** button.



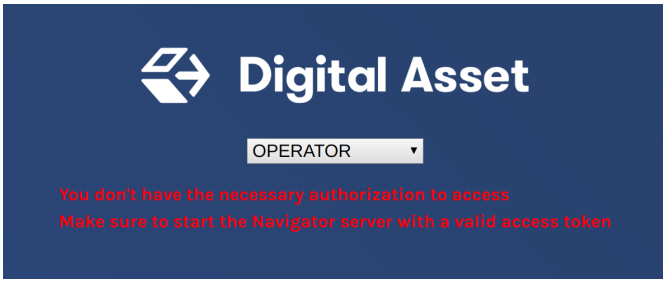
1.15.7.6 Authorizing Navigator

If you are running Navigator against a Ledger API server that verifies authorization, you must provide the access token when you start the Navigator server.

The access token retrieval depends on the specific Daml setup you are working with: please refer to the ledger operator to learn how.

Once you have retrieved your access token, you can provide it to Navigator by storing it in a file and provide the path to it using the `--access-token-file` command line option.

If the access token cannot be retrieved, is missing or wrong, you'll be unable to move past the Navigator's frontend login screen and see the following:



1.15.7.7 Advanced Usage

Customizable table views

Customizable table views is an advanced rapid-prototyping feature, intended for Daml developers who wish to customize the Navigator UI without developing a custom application.

To use customized table views:

1. Create a file `frontend-config.js` in your project root folder (or the folder from which you run Navigator) with the content below:

```
import { DamlLfValue } from '@da/ui-core';

export const version = {
  schema: 'navigator-config',
  major: 2,
  minor: 0,
};

export const customViews = (userId, party, role) => ({
  customview1: {
    type: "table-view",
    title: "Filtered contracts",
    source: {
      type: "contracts",
      filter: [
        {
          field: "id",
          value: "1",
        }
      ],
      search: "",
      sort: [
        {
          field: "id",
          direction: "ASCENDING"
        }
      ]
    },
    columns: [
      {
        key: "id",
        title: "Contract ID",
        createCell: ({rowData}) => ({
          type: "text",
          value: rowData.id
        }),
        sortable: true,
        width: 80,
        weight: 0,
        alignment: "left"
      },
      {
        key: "template.id",
        title: "Template ID",
        createCell: ({rowData}) => ({
```

(continues on next page)

(continued from previous page)

```
          type: "text",
          value: rowData.template.id
        }),
        sortable: true,
        width: 200,
        weight: 3,
        alignment: "left"
      }
    ]
  }
})
```

2. Reload your Navigator browser tab. You should now see a sidebar item titled Filtered contracts that links to a table with contracts filtered and sorted by ID.

To debug config file errors and learn more about the config file API, open the Navigator `/config` page in your browser (e.g., http://localhost:7500/config).

Using Navigator with a Daml Ledger

By default, Navigator is configured to use an unencrypted connection to the ledger. To run Navigator against a secured Daml Ledger, configure TLS certificates using the `--pem`, `--crt`, and `--cacrt` command line parameters. Details of these parameters are explained in the command line help:

```
daml navigator --help
```

1.15.8 Daml Profiler

The Daml Profiler is only available in Daml Enterprise.

The Daml Profiler allows you to to profile execution of your Daml code which can help spot bottlenecks and opportunities for optimization.

1.15.8.1 Usage

To test this out, we use the skeleton project included in the assistant. We first create the project and build the DAR.

```
daml new profile-tutorial --template skeleton
cd profile-tutorial
daml build
```

Next we load the DAR into Sandbox with a special `profile-dir` option. Sandbox will behave as usual but all profile results will be written to that directory. For this, we first create a configuration file that sets the `profile-dir` for Sandbox:

Listing 29: profile.conf

```
canton.participants.sandbox.features.profile-dir = profile-results
```

We then pass

```
daml sandbox --dar .daml/dist/profile-tutorial-0.0.1.dar -c profile.conf
```

To actually produce some profile results, we have to create transactions. For the purposes of this tutorial, the Daml Script included in the skeleton project does the job admirably:

```
daml script --dar .daml/dist/profile-tutorial-0.0.1.dar --ledger-host localhost --
↪ledger-port 6865 --script-name Main:setup
```

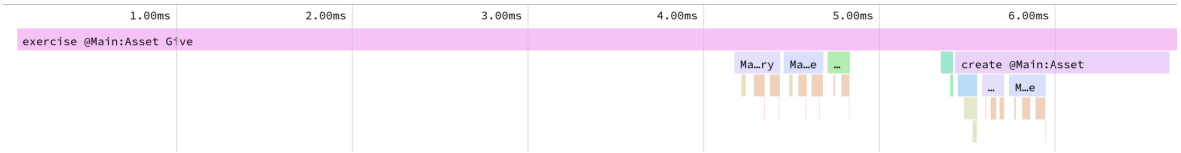
If we now look at the contents of the `profile-results` directory, we can see one JSON file per transaction produced by the script. Each file has a name of the form `$timestamp-$command.json` where `$timestamp` is the submission time of the transaction and `$command` is a human-readable description of the command that produced the transaction (for multi-command submissions, only the first one will be in the file name).

```
$ ls profile-results
2021-03-17T12:32:16.846404Z-create:Asset.json
2021-03-17T12:32:17.361596Z-exercise:Asset:Give.json
2021-03-17T12:32:17.623537Z-exercise:Asset:Give.json
```

At this point, you can stop Sandbox.

To view the profiling results you can use speedscope. The easiest option is to use the web version but you can also install it locally.

Let's open the first exercise profile above `2021-03-17T12:32:17.361596Z-exercise:Asset:Give.json`:



You can see the exercise as the root of the profile. Below that there are a few expressions to calculate signatories, observer and controllers and finally we see the create of the contract. In this simple example, nothing obvious stands out that we could do to optimize further.

Speedscope provides a few other views that can be useful depending on your profile. Refer to the documentation for more information on that.

1.15.8.2 Caveats

1. The profiler currently does not take time into account that is spent outside of pure interpretation, e.g., time needed to fetch a contract from the database.
2. The profiler operates on Daml-LF. This means that the identifiers used in the profiler correspond to Daml-LF expressions which includes autogenerated identifiers used by the compiler. E.g., in the example above, `Main:$csignatory` is the name of the function used to compute signatories of `Asset`. You can view the Daml-LF code that the compiler generated using `daml damlc inspect`. This can be useful to see where an identifier is being used but it does take some experience to be able to read Daml-LF code with ease.

```
daml damlc inspect .daml/dist/profiler-tutorial-0.0.1.dar
```

1.15.9 Daml Codegen

1.15.9.1 Introduction

You can use the Daml codegen to generate Java, and JavaScript/TypeScript classes representing Daml contract templates. These classes incorporate all boilerplate code for constructing corresponding ledger `com.daml.ledger.api.v1.CreateCommand`, `com.daml.ledger.api.v1.ExerciseCommand`, `com.daml.ledger.api.v1.ExerciseByKeyCommand`, and `com.daml.ledger.api.v1.CreateAndExerciseCommand`.

1.15.9.2 Run the Daml Codegen

The basic command to run the Daml codegen is:

```
$ daml codegen [java|js] [options]
```

There are two modes:

Command line configuration, specifying **all** settings in the command line (all codegens supported)
Project file configuration, specifying **all** settings in the `daml.yaml` (currently **Java** only)

Command Line Configuration

Help for each specific codegen:

```
$ daml codegen [java|js] --help
```

Java codegens take the same set of configuration settings:

```
<DAR-file[=package-prefix]>...
                        DAR file to use as input of the codegen with an optional,
↪ but recommend, package prefix for the generated sources.
-o, --output-directory <value>
                        Output directory for the generated sources
-d, --decoderClass <value>
                        Fully Qualified Class Name of the optional Decoder
↪utility
```

(continues on next page)

(continued from previous page)

```
-V, --verbosity <value>  Verbosity between 0 (only show errors) and 4 (show all
↪messages) -- defaults to 0
-r, --root <value>       Regular expression for fully-qualified names of
↪templates to generate -- defaults to .*
--help                   This help text
```

JavaScript/TypeScript codegen takes a different set of configuration settings:

```
DAR-FILES                DAR files to generate TypeScript bindings for
-o DIR                   Output directory for the generated packages
-s SCOPE                 The NPM scope name for the generated packages;
                        defaults to daml.js
-h,--help                Show this help text
```

Project File Configuration (Java)

The above settings can be configured in the `codegen` element of the Daml project file `daml.yaml`. See this issue for status on this feature.

Here is an example:

```
sdk-version: 2.0.0
name: quickstart
source: daml
init-script: Main:initialize
parties:
  - Alice
  - Bob
  - USD_Bank
  - EUR_Bank
version: 0.0.1
exposed-modules:
  - Main
dependencies:
  - daml-prim
  - daml-stdlib
codegen:
  js:
    output-directory: ui/daml.js
    npm-scope: daml.js
  java:
    package-prefix: com.daml.quickstart.iou
    output-directory: java-codegen/src/main/java
    verbosity: 2
```

You can then run the above configuration to generate your **Java** code:

```
$ daml codegen java
```

The equivalent **JavaScript** command line configuration would be:

```
$ daml codegen js ./.daml/dist/quickstart-0.0.1.dar -o ui/daml.js -s daml.js
```

and the equivalent **Java** command line configuration:

```
$ daml codegen java ./.daml/dist/quickstart-0.0.1.dar=com.daml.quickstart.iou --
↪output-directory=java-codegen/src/main/java --verbosity=2
```

In order to compile the resulting **Java** classes, you need to add the corresponding dependencies to your build tools.

For **Java**, add the following **Maven** dependency:

```
<dependency>
  <groupId>com.daml</groupId>
  <artifactId>bindings-java</artifactId>
  <version>YOUR_SDK_VERSION</version>
</dependency>
```

Note: Replace `YOUR_SDK_VERSION` with the version of your SDK

1.16 Daml Finance Documentation

Welcome to the Daml Finance documentation. This page provides an overview of the documentation content as well as suggested starting points. Use the left-hand menu to explore the various sections, or the search bar above for quick navigation. If you are missing content from the documentation, have feedback on the library, or need any help using it, do not hesitate to open an issue on the repository.

1.16.1 Content

Overview: description of the purpose of the library, its high-level architecture, as well as targeted use cases
Concepts: explanation of the main concepts used throughout the library, and how they fit together
Instruments: description of the instruments that are included in Daml Finance and can be used out of the box
Packages: documentation for each individual package and its contained modules
Tutorials: step-by-step implementation guides across different use cases
Reference: glossary as well as code-level documentation for each package

1.16.2 Starting Points

The following is a suggested learning path to get productive quickly:

1. *Get started* quickly
2. Read up on the *background, purpose, and intended usage* of the library
3. Understand the *fundamental concepts* in depth
4. Learn how to *use the instrument packages to model different financial instruments*
5. Explore the *Daml Finance Demo Application*

1.16.3 Releases

This section details the list of released packages for each Daml SDK release. It also provides status information for each package according to the Daml Ecosystem convention.

1.16.3.1 Daml SDK 2.7.0

Stable Packages

Package	Version	Status
ContingentClaims.Core	2.0.0	Stable
ContingentClaims.Lifecycle	2.0.0	Stable
Daml.Finance.Account	2.0.0	Stable
Daml.Finance.Claims	2.0.0	Stable
Daml.Finance.Data	2.0.0	Stable
Daml.Finance.Holding	2.0.0	Stable
Daml.Finance.Instrument.Bond	1.0.0	Stable
Daml.Finance.Instrument.Generic	2.0.0	Stable
Daml.Finance.Instrument.Token	2.0.0	Stable
Daml.Finance.Interface.Account	2.0.0	Stable
Daml.Finance.Interface.Claims	2.0.0	Stable
Daml.Finance.Interface.Data	3.0.0	Stable
Daml.Finance.Interface.Holding	2.0.0	Stable
Daml.Finance.Interface.Instrument.Base	2.0.0	Stable
Daml.Finance.Interface.Instrument.Bond	1.0.0	Stable
Daml.Finance.Interface.Instrument.Generic	2.0.0	Stable
Daml.Finance.Interface.Instrument.Token	2.0.0	Stable
Daml.Finance.Interface.Lifecycle	2.0.0	Stable
Daml.Finance.Interface.Settlement	2.0.0	Stable
Daml.Finance.Interface.Types.Common	1.0.1	Stable
Daml.Finance.Interface.Types.Date	2.0.1	Stable
Daml.Finance.Interface.Util	2.0.0	Stable
Daml.Finance.Lifecycle	2.0.0	Stable
Daml.Finance.Settlement	2.0.0	Stable
Daml.Finance.Util	3.0.0	Stable

Early Access Packages

Package	Version	Status
ContingentClaims.Valuation	0.2.1	Labs
Daml.Finance.Instrument.Equity	0.3.0	Alpha
Daml.Finance.Instrument.Option	0.2.0	Alpha
Daml.Finance.Instrument.Swap	0.3.0	Alpha
Daml.Finance.Interface.Instrument.Equity	0.3.0	Alpha
Daml.Finance.Interface.Instrument.Option	0.2.0	Alpha
Daml.Finance.Interface.Instrument.Swap	0.3.0	Alpha

Deprecated Packages

Package	Version	Status
ContingentClaims.Core	1.*	Depr.
ContingentClaims.Lifecycle	1.*	Depr.
Daml.Finance.Account	1.*	Depr.
Daml.Finance.Claims	1.*	Depr.
Daml.Finance.Data	1.*	Depr.
Daml.Finance.Holding	1.*	Depr.
Daml.Finance.Instrument.Generic	1.*	Depr.
Daml.Finance.Instrument.Token	1.*	Depr.
Daml.Finance.Interface.Account	1.*	Depr.
Daml.Finance.Interface.Claims	1.*	Depr.
Daml.Finance.Interface.Data	2.*	Depr.
Daml.Finance.Interface.Holding	1.*	Depr.
Daml.Finance.Interface.Instrument.Base	1.*	Depr.
Daml.Finance.Interface.Instrument.Generic	1.*	Depr.
Daml.Finance.Interface.Instrument.Token	1.*	Depr.
Daml.Finance.Interface.Lifecycle	1.*	Depr.
Daml.Finance.Interface.Settlement	1.*	Depr.
Daml.Finance.Interface.Util	1.*	Depr.
Daml.Finance.Lifecycle	1.*	Depr.
Daml.Finance.Settlement	1.*	Depr.
Daml.Finance.Util	2.*	Depr.

1.16.3.2 Daml SDK 2.6.0

Stable Packages

Package	Version	Status
Daml.Finance.Account	1.0.1	Stable
Daml.Finance.Claims	1.0.1	Stable
Daml.Finance.Data	1.0.1	Stable
Daml.Finance.Holding	1.0.2	Stable
Daml.Finance.Instrument.Generic	1.0.1	Stable
Daml.Finance.Instrument.Token	1.0.1	Stable
Daml.Finance.Interface.Data	2.0.0	Stable
Daml.Finance.Interface.Types.Date	2.0.0	Stable
Daml.Finance.Lifecycle	1.0.1	Stable
Daml.Finance.Settlement	1.0.2	Stable
Daml.Finance.Util	2.0.0	Stable

Early Access Packages

Package	Version	Status
Daml.Finance.Instrument.Bond	0.2.1	Alpha
Daml.Finance.Instrument.Equity	0.2.1	Alpha
Daml.Finance.Instrument.Option	0.1.0	Alpha
Daml.Finance.Instrument.Swap	0.2.1	Alpha
Daml.Finance.Interface.Instrument.Bond	0.2.1	Alpha
Daml.Finance.Interface.Instrument.Option	0.1.0	Alpha
Daml.Finance.Interface.Instrument.Swap	0.2.1	Alpha

Deprecated Packages

Package	Version	Status
Daml.Finance.Interface.Data	1.*	Depr.
Daml.Finance.Interface.Types.Date	1.*	Depr.
Daml.Finance.Util	1.*	Depr.

1.16.3.3 Daml SDK 2.5.0

Stable Packages

Package	Version	Status
ContingentClaims.Core	1.0.0	Stable
ContingentClaims.Lifecycle	1.0.0	Stable
Daml.Finance.Account	1.0.0	Stable
Daml.Finance.Claims	1.0.0	Stable
Daml.Finance.Data	1.0.0	Stable
Daml.Finance.Holding	1.0.1	Stable
Daml.Finance.Instrument.Generic	1.0.0	Stable
Daml.Finance.Instrument.Token	1.0.0	Stable
Daml.Finance.Interface.Account	1.0.0	Stable
Daml.Finance.Interface.Claims	1.0.0	Stable
Daml.Finance.Interface.Data	1.0.0	Stable
Daml.Finance.Interface.Holding	1.0.0	Stable
Daml.Finance.Interface.Instrument.Base	1.0.0	Stable
Daml.Finance.Interface.Instrument.Generic	1.0.0	Stable
Daml.Finance.Interface.Instrument.Token	1.0.0	Stable
Daml.Finance.Interface.Lifecycle	1.0.0	Stable
Daml.Finance.Interface.Settlement	1.0.0	Stable
Daml.Finance.Interface.Types.Common	1.0.0	Stable
Daml.Finance.Interface.Types.Date	1.0.0	Stable
Daml.Finance.Interface.Util	1.0.0	Stable
Daml.Finance.Lifecycle	1.0.0	Stable
Daml.Finance.Settlement	1.0.1	Stable
Daml.Finance.Util	1.0.0	Stable

Early Access Packages

Package	Version	Status
ContingentClaims.Valuation	0.2.0	Labs
Daml.Finance.Instrument.Bond	0.2.0	Alpha
Daml.Finance.Instrument.Equity	0.2.0	Alpha
Daml.Finance.Instrument.Option	0.1.0	Alpha
Daml.Finance.Instrument.Swap	0.2.0	Alpha
Daml.Finance.Interface.Instrument.Bond	0.2.0	Alpha
Daml.Finance.Interface.Instrument.Equity	0.2.0	Alpha
Daml.Finance.Interface.Instrument.Option	0.1.0	Alpha
Daml.Finance.Interface.Instrument.Swap	0.2.0	Alpha

Deprecated Packages

Package	Version	Status
None		

1.17 Overview

This overview section describes the purpose of the Daml Finance library, its high-level architecture, as well as targeted use cases.

1.17.1 Introduction

1.17.1.1 Purpose

Daml Finance supports the modeling of financial and non-financial use cases in Daml. It provides a standard way to represent assets on Daml ledgers and defines common behaviours and rules. There are two main benefits to using the library in your application:

Shortened time-to-market
: Implementing basic financial concepts like ownership or economic terms of an asset is a complex and tedious task. By providing common building blocks, Daml Finance increases delivery velocity and shortens the time-to-market when building Daml applications. The rich set of functionality of Daml Finance is at your disposal so you don't have to reinvent the wheel.

Application composability
: Building your application on Daml Finance makes it compatible with other platforms in the wider ecosystem. By using a shared library assets become mobile , allowing them to be used seamlessly across application boundaries without the need for translation or integration layers. For instance, a Daml Finance-based asset that is originated in a bond issuance application can be used in the context of a secondary market trading application that is also built on Daml Finance.

1.17.1.2 Design Goals

Daml Finance optimizes for the following aspects:

Accessibility
: The library is designed to have a low barrier to entry. Users familiar with Daml can get started quickly and leverage the provided functionality easily.

Maintainability
: Building with Daml Finance decouples your application code from the underlying representation of assets. This allows the application to evolve without the need to migrate assets from one version to another, and makes maintenance easier.

Extensibility
: Various extension points allow customization and extension of the library as required. If an existing implementation does not fulfill the requirements it is straightforward to provide a custom extension.

1.17.1.3 Scope

The library covers the following areas:

Holdings: modeling of ownership structures, custodial relationships, intermediated securities, and accounts
Instruments: structuring the economic terms of an asset and the events that govern its evolution
Settlement: executing complex transactions involving multiple parties and assets
Lifecycling: governing the evolution of financial instruments over their lifetime

1.17.1.4 Use Cases

Daml Finance comes with broad asset and workflow capabilities to allow for a variety of use cases to be modeled:

Simple tokens: digital representation of traditional assets
Central bank digital currency: retail or wholesale distribution models
Standard asset classes: equities with corporate actions, bonds with flexible cash flow modeling
Derivatives: time- and path-dependent derivatives with optionality
Synchronized lifecycling: atomic, intermediated lifecycling and settlement of cash flows across investors and custodians
Cross-entity issuance: atomic, multi-party issuance across investors, issuer, risk book, and treasury
Asset-agnostic trading facility: generic delivery-vs-payment and immediate, guaranteed settlement
Exotic asset types: non-fungible and non-transferable assets

1.17.1.5 Exploring the Library

If you want to review the Daml Finance codebase in more detail you can clone the repository locally on your machine. This allows you to navigate the code, including both the template definitions and the tests. In particular the tests are useful to show how the library works and how the different components interact with each other. If you need to view the code for a specific package release, you can check out the corresponding tag.

As a pre-requisite, the Daml SDK needs to be installed on your machine.

In order to download the repository, open a terminal and run:

```
git clone git@github.com:digital-asset/daml-finance.git
```

This creates a new folder `daml-finance` containing the Daml Finance source code. Navigate to the folder and run:

```
make build
```

This downloads all required packages and builds the project. You can then run:

```
daml studio
```

to open the code editor and inspect the code.

1.17.1.6 Demo Application

In addition to Daml Finance, there is also a separate Demo Application, showcasing several of the library's capabilities in a web-based graphical user interface.

If you are interested in trying out the app locally, you can clone the corresponding repo and follow the installation instructions on the Daml Finance Demo App GitHub page.

1.17.2 Architecture

This page outlines the architecture of the library and the relationships between the different packages.

Daml Finance consists of a set of `.dar` packages that can be divided into two layers:

an *interface layer* representing its public, stable API
an *implementation layer* providing a set of default implementation packages

1.17.2.1 Interface Layer

The interface layer provides common types and Daml interface definitions that represent the public API of Daml Finance. It includes several Daml packages, each grouping related business functions. These packages can in principle be used independently of each other.

The interface layer consists of the following packages:

`Daml.Finance.Interface.Holding` defines interfaces for holdings and related properties such as *transferability* or *fungibility*.
`Daml.Finance.Interface.Account` defines interfaces for accounts
`Daml.Finance.Interface.Settlement` defines interfaces for settlement route providers, settlement instructions, and batched settlements
`Daml.Finance.Interface.Lifecycle` defines interfaces used for instrument lifecycling
`Daml.Finance.Interface.Instrument.*` contains interfaces used for different instrument types
`Daml.Finance.Interface.Claims` contains interfaces used for *Contingent Claims* based instrument types
`Daml.Finance.Interface.Data` defines interfaces related to reference data
`Daml.Finance.Interface.Types.Common` provides common types
`Daml.Finance.Interface.Types.Date` provides types related to dates
`Daml.Finance.Interface.Util` defines utilities and interfaces used by other interface packages.
`ContingentClaims.Core` contains types for representing *Contingent Claims* tree structures.

1.17.2.2 Implementation Layer

The implementation layer contains concrete template definitions implementing the interfaces defined in the interface layer. These represent the contracts that are ultimately stored on the ledger.

For instance, `Daml.Finance.Holding` contains a concrete implementation of a *Transferable* and *Fungible* holding. These interfaces are defined in `Daml.Finance.Interface.Holding`.

The implementation layer consists of the following packages:

`Daml.Finance.Holding` defines default implementations for holdings
`Daml.Finance.Account` defines default implementations for accounts
`Daml.Finance.Settlement` defines templates for settlement route providers, settlement instructions, and batched settlements
`Daml.Finance.Lifecycle` defines an implementation of lifecycle effects and a rule template to facilitate their settlement
`Daml.Finance.Instrument.*` contains implementations for various instrument types
`Daml.Finance.Data` includes templates used to store reference data on the ledger
`Daml.Finance.Claims` contains utility functions relating to *Contingent Claims* based instruments and lifecycling
`Daml.Finance.Util` provides a set of pure utility functions mainly for date manipulation
`ContingentClaims.Lifecycle` provides lifecycle utility functions for *Contingent Claims* based instruments
`ContingentClaims.Valuation` contains experimental functions to transform *Contingent Claims* instrument trees into a mathematical representation suitable for integration with pricing and risk frameworks

1.17.2.3 Versioning and Compatibility

Daml Finance follows the semantic versioning scheme.

The interface packages define the public API of the library. Specifically, the interface definitions which include interface views, methods and choices are guaranteed to remain stable within a major version of a package. Note that this does not include the package id itself. So purely additive (e.g. adding new interfaces), or non-functional changes (like compiling a package with a later SDK version), which do change the package id of a package but do not change the interface definitions, can be released in minor or patch version increments. Such changes will require dependent applications to be recompiled and upgraded, but the upgrades are trivial as none of the existing interfaces changed functionally.

Implementation packages follow a similar convention. A purely additive change, or a change that does not affect the implemented interfaces can be rolled out as a minor or patch version increase. Similarly, an upgrade to implement a new *minor or patch* version of of an interface, which doesn't functionally change the interface implementation is also considered a minor or patch version increase of an implementation package. If an implementation package changes to implement a new major version of an interface the major version of the implementation will change as well.

We intend to document the upgrade process and/or provide sample upgrade scripts for contracts within the Daml Finance perimeter for major version upgrades only.

Note that deprecations of package versions only happen in the context of a Daml SDK release. They will be listed in the *release section* of the documentation and follow the standard Daml component deprecation guidelines.

1.17.3 Building Applications

This page describes the patterns to follow when building applications using Daml Finance.

1.17.3.1 Installing Daml Finance

Each Daml SDK release defines a set of consistent Daml Finance package versions that have been tested to work with each other. The list of package versions for each Daml SDK release can be found *here*. To facilitate getting started with a particular release set, the Daml SDK comes with a `quickstart-finance` template that contains a script to download these packages.

After installing the Daml SDK, you can execute the following commands to create a new Daml Finance project based on the set of packages released with the given SDK version:

On Unix-based systems execute:

```
daml new quickstart-finance --template=quickstart-finance
cd quickstart-finance
./get-dependencies.sh
```

On Windows-based systems execute:

```
daml new quickstart-finance --template=quickstart-finance
cd quickstart-finance
get-dependencies.bat
```

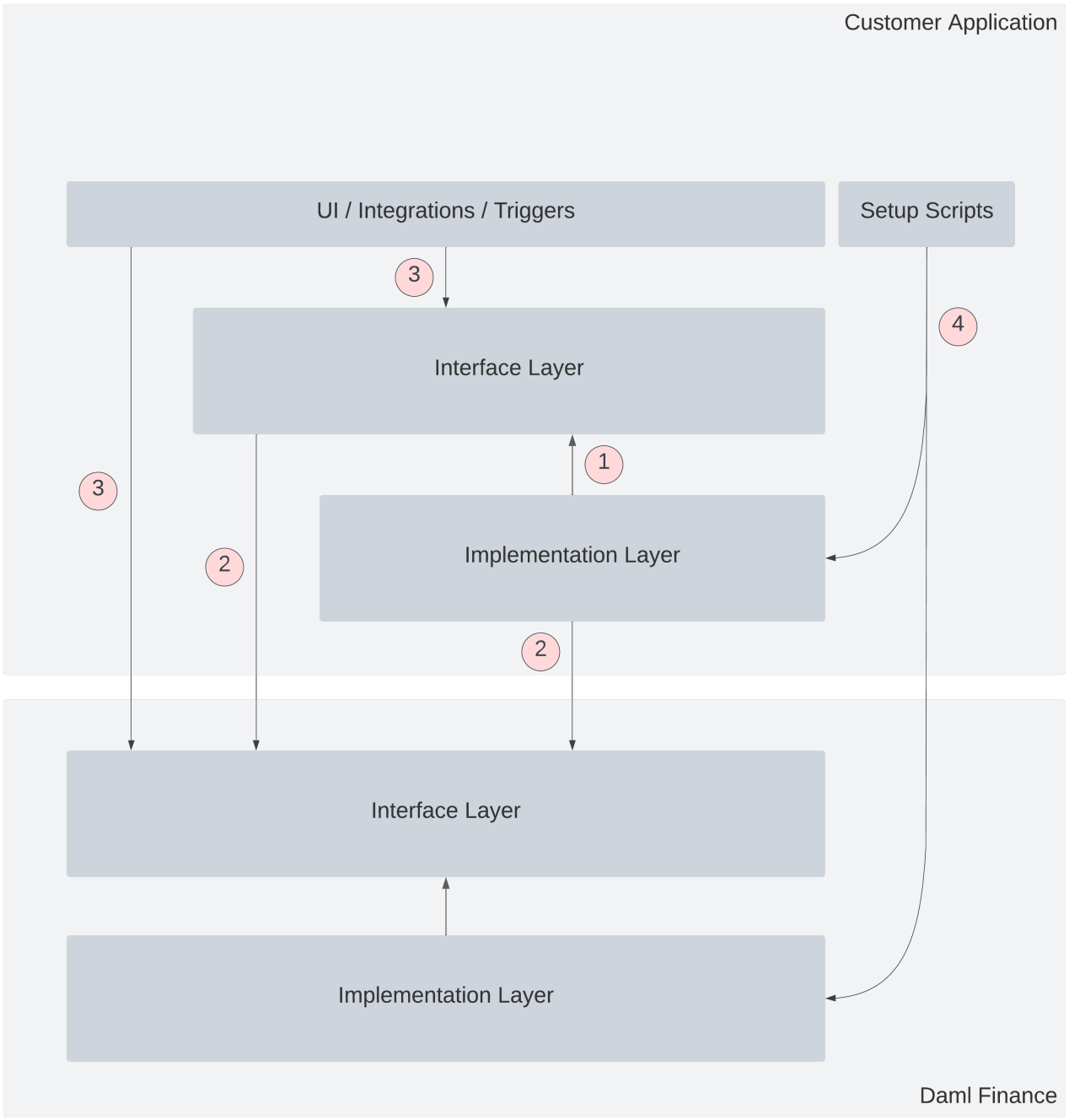
You can then edit the `daml.yaml` file and uncomment the lines corresponding to the packages you require in your project.

Alternatively, if you want to install the latest Daml Finance version into an existing project, you can copy and execute these scripts (Unix and Windows variants) from the main branch of the repository.

1.17.3.2 Application Architecture

When building applications using Daml Finance it is important to ensure your application only depends on the interface layer (i.e., the public API) of Daml Finance. Furthermore, it is suggested that your application follows a similar split between interface (API) and implementation layer in order to maximize upgradability and minimize the impact of incremental changes to your application or Daml Finance.

The following picture shows a suggested architecture that minimizes undesirable coupling and optimizes for upgradability of your application:



The following annotations are highlighted in the diagram:

1. The customer application should be split into an implementation and an interface (API) layer. This ensures that implementations can be upgraded without affecting client-side applications, like the UI, integrations, or Daml Triggers.
2. The customer application (both the interface and implementation layer) should only depend on the interface layer (API) of Daml Finance. This ensures that upgrades to the implementation layer of Daml Finance do not affect the Customer Application.
3. All client-side code (UI, integrations, Daml Triggers, etc.) should only depend on the interface layers of Daml Finance and the Customer Application. This ensures that any implementation upgrades in Daml Finance or the Customer Application do not affect client-side code.
4. Any setup scripts used to initialize the application can (and usually have to) depend on the implementation layers of Daml Finance and the Customer Application. This is required to set

up contracts like factories, where a dependency on the implementation package is needed. It does not affect the overall upgradability of the Customer Application as these operations are usually executed either at initial setup or on a one-off basis, but not during normal operation of the application.

Following the above patterns ensures minimal impact of changes to any implementation part of the overall application:

If a Daml Finance implementation package is upgraded, only the contracts for templates within the package have to be upgraded. The Customer Application itself is unaffected because it only depends on the interface packages, which remain unchanged.
If a customer application implementation package is upgraded, only the contracts for templates within that package need to be upgraded. The client-side code of the application is unaffected, as it only depends on the Customer Application interface layer.
If a Daml Finance interface package is upgraded, the affected parts in the customer application implementation, interface, or client-side layer need to be upgraded. To minimize the impact of such change it is suggested that the customer application layers themselves are divided into packages, that each depend on a minimal set of Daml Finance interface packages.
If a customer application interface package is upgraded the corresponding implementation packages, as well as the affected client-side code have to be upgraded. Again, splitting up the interface (API) layer of the customer application can minimize the impact of such a change.

In general, we will provide upgrade contracts and scripts to facilitate migration between major version updates of packages within the Daml Finance perimeter.

1.17.3.3 Using Daml Codegen

The Daml Finance packages are compatible with the *Daml Codegen tool*.

If you, e.g., want to create a *JavaScript* app that uses Daml Finance, it is possible to generate *JavaScript* classes from the Daml Finance packages you need. Use *daml codegen js*, for example:

```
daml codegen js -o ./output .lib/daml-finance-interface-instrument-swap-0.2.1.dar
↪.lib/daml-finance-interface-instrument-bond-0.2.1.dar
```

Alternatively, if your app uses *Java*, you can run *daml codegen java* in a similar way:

```
daml codegen java -o ./output .lib/daml-finance-interface-instrument-swap-0.2.1.
↪dar .lib/daml-finance-interface-instrument-bond-0.2.1.dar
```

Note, this Daml Finance codegen is only supported on SDK versions 2.5.x and higher.

1.17.4 Extending Daml Finance

Daml Finance is designed to be extended whenever the provided implementations do not satisfy the requirements at hand. In principle, all interfaces in the *interface layer* can be implemented with custom implementations. Specific extension points we expect and encourage users to customize are explained below.

Note that for all of the listed extension points, we are happy to receive external contributions to be included in the library.

1.17.4.1 Custom Holding Implementations

Daml Finance provides default implementations for fungible, non-fungible, and non-transferable holdings. The transferability of transferable holdings can be flexibly controlled through the *controllers* property on an *Account*. Some use cases, however, might require additional functionality on holding contracts:

Restricted transferability: a custom implementation of the *Transferable interface* can enforce additional conditions (e.g. the presence of some contract) required to transfer a holding.
Fixed divisibility: a custom implementation of the *Fungible interface* can enforce specific requirements regarding the divisibility of a holding.
Additional information: a custom implementation of a holding can provide additional information, for example, the timestamp of when the holding was obtained. This can be used to implement features that depend on the time a particular asset has been held (e.g. holding fees, interest, etc.).

Note that any custom holding implementation will still allow you to leverage other parts of the library (e.g. lifecycling or settlement) as those are implemented against the respective interfaces. You will need to provide an implementation of the *Holding Factory* interface for your implementation to be usable throughout the library.

1.17.4.2 Custom Account Implementations

The default account implementation in Daml Finance allows you to define authorization requirements for incoming and outgoing transfers through the *controllers* property. For some cases, however, a custom account implementation may be warranted:

Restricted credit and debit: a custom implementation of the `Credit` and / or `Debit` choices on the *Account interface* can place additional restrictions on those actions that can depend, for example, on the presence of a separate know-your-customer (KYC) contract.
Additional information: a custom account implementation can serve to represent different concepts of accounts. For example, a shelf in a vault for gold bars or a specific location within a warehouse can be represented by providing additional information on an account implementation.

1.17.4.3 Custom Instrument Implementations

Daml Finance provides default implementations for a wide range of financial instruments. However, we anticipate that specific requirements will lead to the adaptation of existing, or the creation of entirely new instrument types. The following are typical examples of when a custom instrument implementation is required:

Additional information: a custom instrument implementation might, for example, build upon the *Equity interface* to provide additional information pertinent to private equity (like share class, or liquidation preference).
New instrument types: if Daml Finance does not provide an implementation for a given instrument type, a custom implementation can be provided to fill that gap. The implementation can either leverage the *Contingent Claims* framework, as described in *this tutorial*, or be implemented through standard interfaces, as seen in the implementation of the *Equity instrument*.

1.17.4.4 Custom Lifecycle Implementations

Daml Finance provides a default set of lifecycle rules that can be used to evolve instruments. Examples are the implementation of *Distributions*, *Replacements*, or the *time-based evolution* of contingent-claims based instruments. There are many more lifecycle events and rules that can be implemented using the provided interfaces. Typically, implementations of the *Event* and *Rule* interface are required to handle new lifecycle events. Examples of events where a library extension might be warranted include:

Credit events on bonds: our bond implementations don't provide an implementation for handling default events, as these are highly case-specific. A custom lifecycle event and rule implementation can provide the logic to handle the treatment of bond positions in case of default.
Special corporate actions: a distribution that is either restricted to, or dependent on certain conditions can be implemented through a custom lifecycle implementation.
Custom evolution logic: a non-fungible token following a specific evolution logic (i.e., it can be evolved under certain circumstances) can be implemented using custom lifecycle events and rules.

1.17.4.5 Custom Settlement Implementations

Daml Finance aims to provide a flexible and powerful mechanism to orchestrate asset settlement. There are cases, however, where a custom implementation might be required:

Off-ledger integrations: specific information might be required to facilitate handling of settlement instructions in off-ledger rails. This could include, for example, information required to create SWIFT messages.
Cross-ledger settlement: mechanisms like Hashed Timelock Contracts or custodial-bridged settlement might require a custom implementation of the settlement choices.

1.18 Concepts

This section describes the core concepts of the Daml Finance library. It also refers the reader to where each of these concepts is implemented in the library.

The most important definitions are also summarized in the *glossary*.

1.18.1 Asset Model

The library's asset model is the set of contracts that describe the financial rights and obligations that exist between parties. It is composed of instruments, holdings, and accounts.

1.18.1.1 Instrument

An instrument contract describes the economic terms (rights and obligations) of one unit of a financial contract.

It can be as simple as an ISIN code referencing some real-world (off-ledger) security, or it can encode specific on-ledger lifecycling logic.

Signatories

Every instrument must have an `issuer` party and a `depository` party, which are both signatories of the contract.

The terminology is borrowed from the real world. For example, an issuer of a stock instrument deposits the paper certificate at a depository and gets the corresponding amount credited in book-entry form.

On the ledger, the `depository` acts as a trusted party that prevents the `issuer` from potentially acting maliciously.

Keys and Versioning

Instruments are keyed by an *InstrumentKey*, which comprises:

the instrument `issuer`
the instrument `depository`
a textual `id`
a textual `version`

The version is used to keep track of the linear evolution of an instrument. For example, once a dividend on a share is paid, the version is used to identify the cum-dividend and the ex-dividend share.

Interfaces

Instrument interfaces are defined in the `Daml.Finance.Interface.Instrument.*` packages.

All instruments must implement the base interface, defined in *Daml.Finance.Interface.Instrument.Base*.

Implementations

A base implementation is provided in *Daml.Finance.Instrument.Token*.

This template does not define any lifecycling logic and is suitable to model contracts that are likely to stay stable, such as currency instruments.

The extension packages provide additional business-specific implementations, such as an *Equity* instrument (where the issuer can pay dividends) or a *Bond* instrument (which includes coupon payments).

The expectation is that customers define their own instruments suiting the use-case they are modeling.

1.18.1.2 Holding

A holding contract represents the ownership of a certain amount of an instrument by an owner at a custodian.

Whereas an instrument defines *what* a party holds (the rights and obligations), a holding defines *how much* (ie., the amount) of an instrument and *against which party* (ie., the custodian) the instrument is being held.

It is important to understand that the economic terms of an asset (the instrument) are separated from the representation of an asset holding. This allows centralized management of instruments (e.g. lifecycling) and the reuse of instruments and associated logic across different entities (e.g. custodians). It also avoids the data redundancy of replicating instrument data and logic on every holding contract.

Signatories

Every holding must have an `owner` party and a `custodian` party, which are usually both signatories of the contract.

The terminology is again borrowed from the real world: our cash or shares are usually deposited at a custodian and we have (at least in principle) the right to claim them back from the custodian at any given time.

Properties of Holdings

A holding implementation can have specific properties such as being *fungible* or *transferable*.

When, for instance, a holding is transferable, the ownership can be transferred to a different party at the same custodian.

These properties are exposed by implementing the corresponding interface (*Fungible* and *Transferable*, respectively).

Interfaces

Holding interfaces are defined in the `Daml.Finance.Interface.Holding` package. These include a *base holding interface*, as well as interface definitions for the above properties.

Implementations

Implementations are provided in `Daml.Finance.Holding` for:

a *fungible and transferable* holding
a holding which is *transferable but not fungible*
a holding which is *neither transferable nor fungible*

1.18.1.3 Account

Account contracts are used as proof of a relationship between a `custodian` and an `owner`.

An `owner` must have an account contract with a `custodian` before a holding contract can be created between the two parties.

This is similar to how, in the real world, you need to open a bank account before you can use the bank's services.

The account contract also controls which parties are authorized to transfer holdings in and out of the account. To be more precise, the *controllers* field of the account contains:

`outgoing`: a set of parties authorizing outgoing transfers
`incoming`: a set of parties authorizing incoming transfers

This allows for modeling various controllers of transfers between Alice's and Bob's accounts. For example:

owners-controlled: If the `owner` is the sole member the `outgoing` and `incoming` controllers for the accounts, a transfer of a holding from Alice's account to Bob's account needs to be authorized jointly by Alice and Bob.
owner-only-controlled: If, instead, there are no `incoming` controllers of Bob's account, it is enough that Alice authorizes the transfer alone.
custodian-controlled: If, as often is the case, the `custodian` needs to control what is being transferred, we can instead let the `custodian` be the sole member of `outgoing` and `incoming` controllers of the accounts.

Accounts also serve to prevent holding transfers to unvetted third parties: a holding of Alice can only be transferred to Bob if Bob has an account at the same Bank (and has therefore been vetted by the Bank).

Signatories

An account is co-signed by the account `owner` and the `custodian`.

Keys

Accounts are keyed by an *AccountKey*, which comprises:

the account `owner`
the account `custodian`
a textual `id`

Interfaces

The account interface is defined in the *Daml.Finance.Interface.Account* package.

Implementations

A base account implementation is provided in *Daml.Finance.Account*.

The account can be created with arbitrary *controllers* (for incoming and outgoing transfers).

In our examples, we typically let accounts be owners-controlled, i.e., both the current owner and the new owner must authorize transfers.

1.18.1.4 Example setups

We can now look at a few examples of how real-world rights and obligations can be modeled using the Daml Finance asset model.

Currency

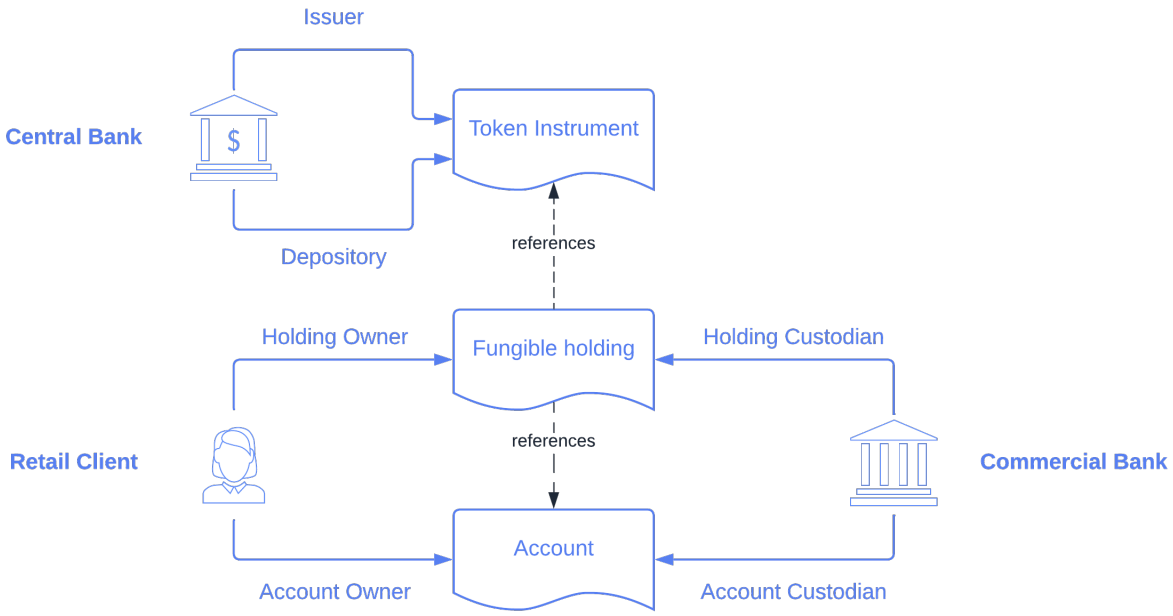
We start by modeling a standard cash bank account. There are three parties involved: a Central Bank, a Commercial Bank, and a Retail Client.

The Central Bank defines the economic terms of the currency asset and is generally a highly trusted entity, therefore it acts as `issuer` as well as `depository` of the corresponding instrument.

We can use the *Token* instrument implementation for a currency asset, as we do not need any lifecycling logic.

The Retail Client has an *Account* at the Commercial Bank, with the former acting as `owner` and the latter as `custodian`.

Finally, the Retail Client is `owner` of a *fungible holding* at the Commercial Bank (the `custodian` in the contract). The holding references the currency instrument, as well as the account.



In this scenario, we can see how:

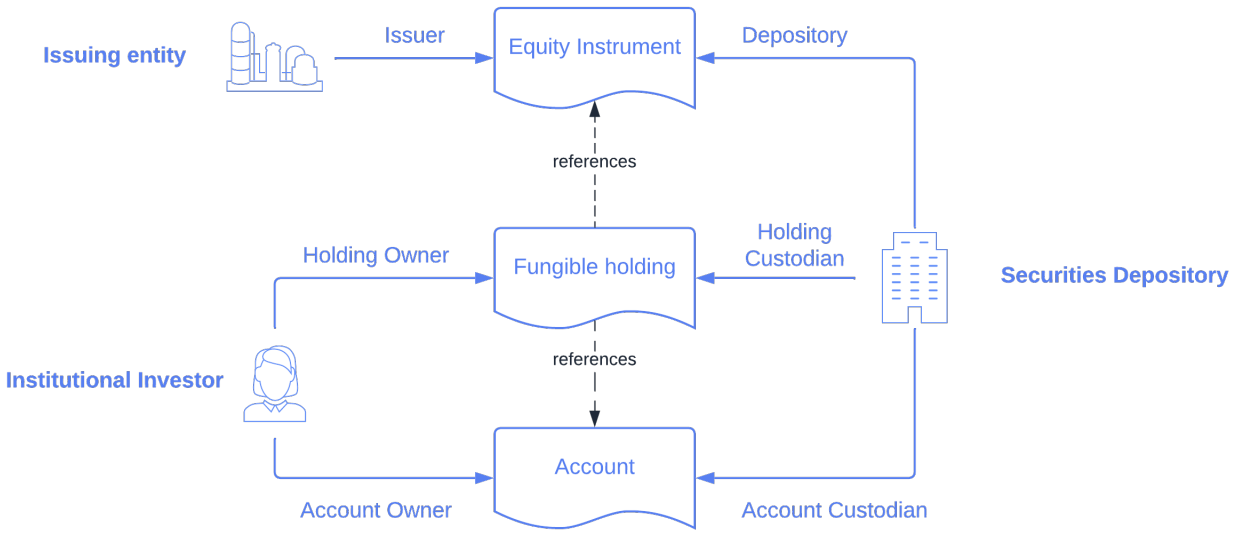
the instrument defines what is held
the holding defines where the rights and obligations lie, as well as the corresponding amount

Equity

We now model units of shares held by an investor. There are three parties involved: an Issuing Entity, a Securities Depository, and an Investor.

The Issuing Entity acts as `issuer` of the *Equity Instrument*. The Securities Depository acts as `depository` of the instrument, thus preventing the Issuing Entity from single-handedly modifying details of the instrument (such as the share's nominal value).

The Institutional Investor holds units of shares against the Securities Depository, through corresponding Account and Holding contracts.

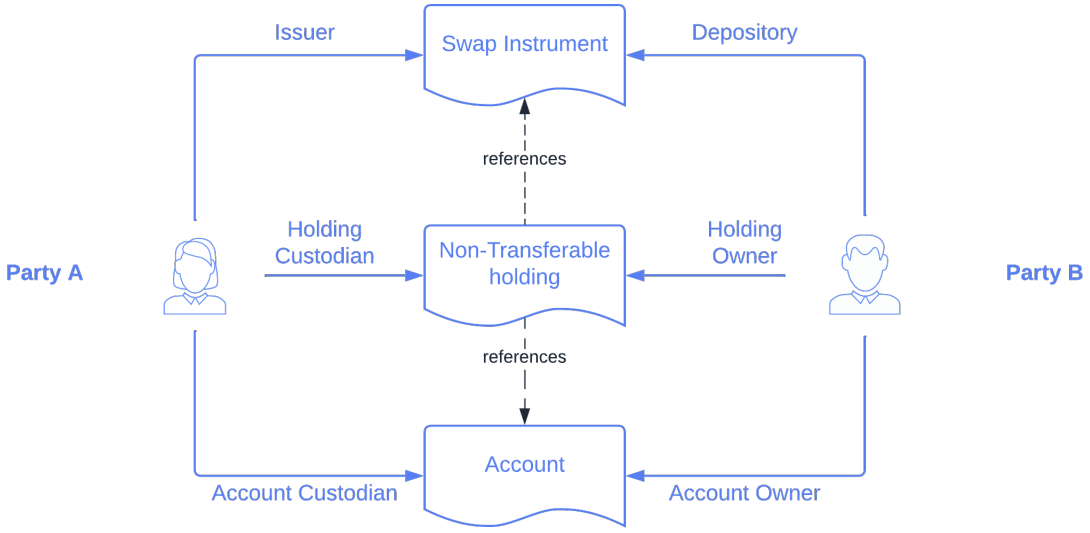


It is worth noting that the `issuer` of the Equity Instrument has the right to perform certain Corporate Actions, such as declaring dividends. This topic is covered in the *lifecycling section*.

OTC Swap

Finally, we model an OTC (over-the-counter) fixed vs. floating interest rate swap agreement between two parties, namely Party A and Party B. We can use the *Interest Rate Swap* instrument template for this purpose.

In this case, all contracts are agreed and co-signed by both parties. In the instrument contract, it does not really matter whether Party A is the `issuer` and Party B the `depository`, or the other way around. However, the role matters in the Holding contract, as it defines the direction of the trade, i.e., which party receives the fixed leg and which party receives the floating one.



1.18.2 Settlement

Settlement refers to the execution of holding transfers originating from a financial transaction.

Daml Finance provides facilities to execute these transfers atomically (i.e., within the same Daml transaction). Interfaces are defined in the `Daml.Finance.Interface.Settlement` package, whereas implementations are provided in the `Daml.Finance.Settlement` package.

In this section, we first illustrate the settlement workflow with the help of an example FX transaction, where Alice transfers a EUR-denominated holding to Bob, in exchange for a USD-denominated holding of the same amount.

We then delve into the details of each of the settlement components.

1.18.2.1 Workflow

Our initial state looks as follows:

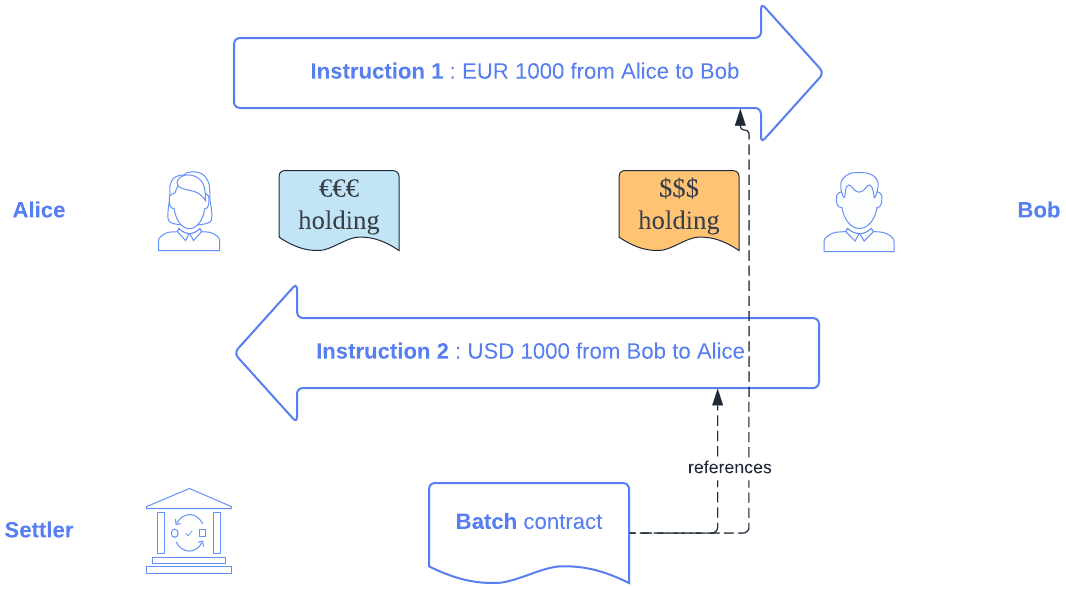
> Alice owns a holding on a `EUR` instrument, for an amount of `1000`
> Bob owns a holding on a `USD` instrument, for an amount of `1000`

These holdings are generally held at different custodians.

Instruct

Alice and Bob want to exchange their holdings and agree to enter into the transaction by being signatories on a transaction contract. Settlement can then be instructed which results in 3 contract instances being created:

1. an *Instruction* to transfer EUR 1000 from Alice to Bob
2. an *Instruction* to transfer USD 1000 from Bob to Alice
3. a *Batch* used to execute the above Instructions



Each instruction defines who is the sender, who is the receiver, and what should be transferred (instrument and amount) at which custodian.

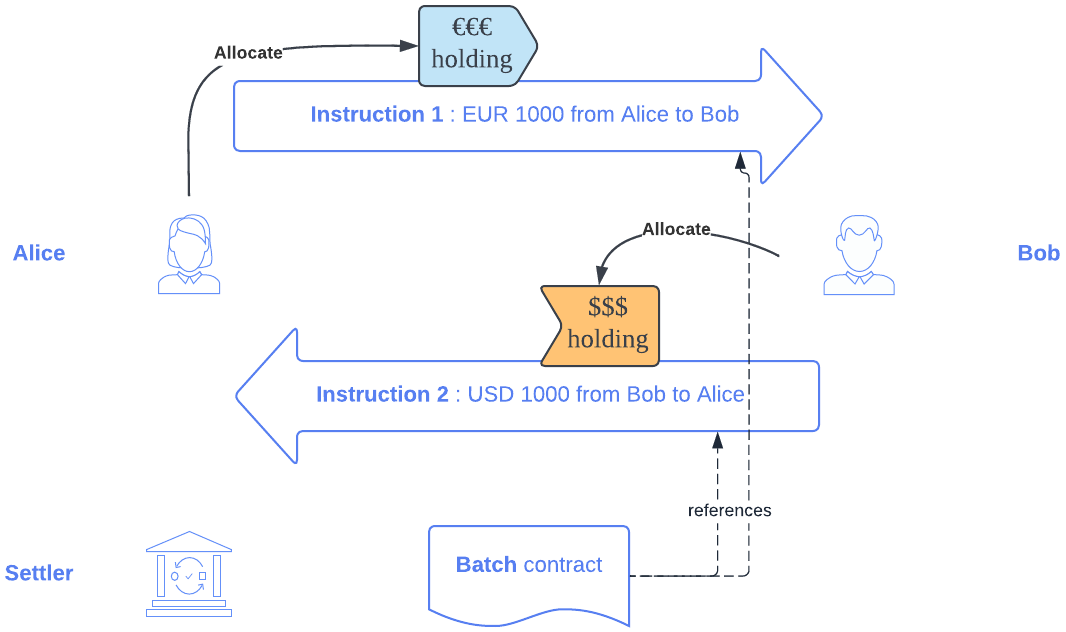
Allocate and Approve

In order to execute the FX transaction, we first need to:

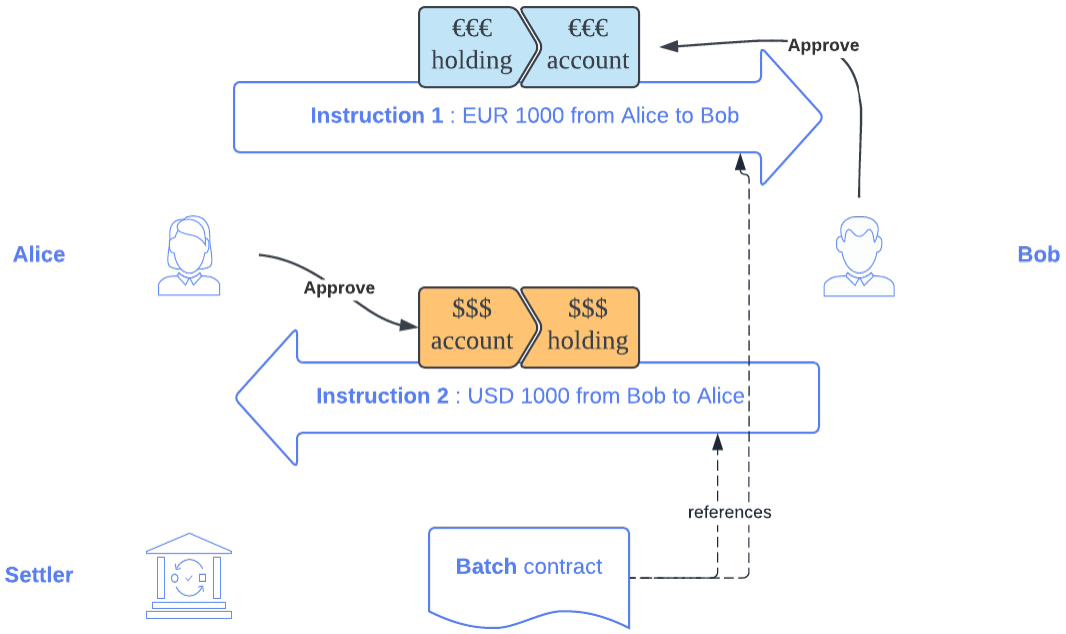
allocate, i.e., specify which holding should be used
approve, i.e., specify to which account the asset should be transferred

Allocation and approval is required for each *Instruction*.

Alice *allocates* the instruction where she is the sender by pledging her holding. Bob does the same on the instruction where he is the sender.

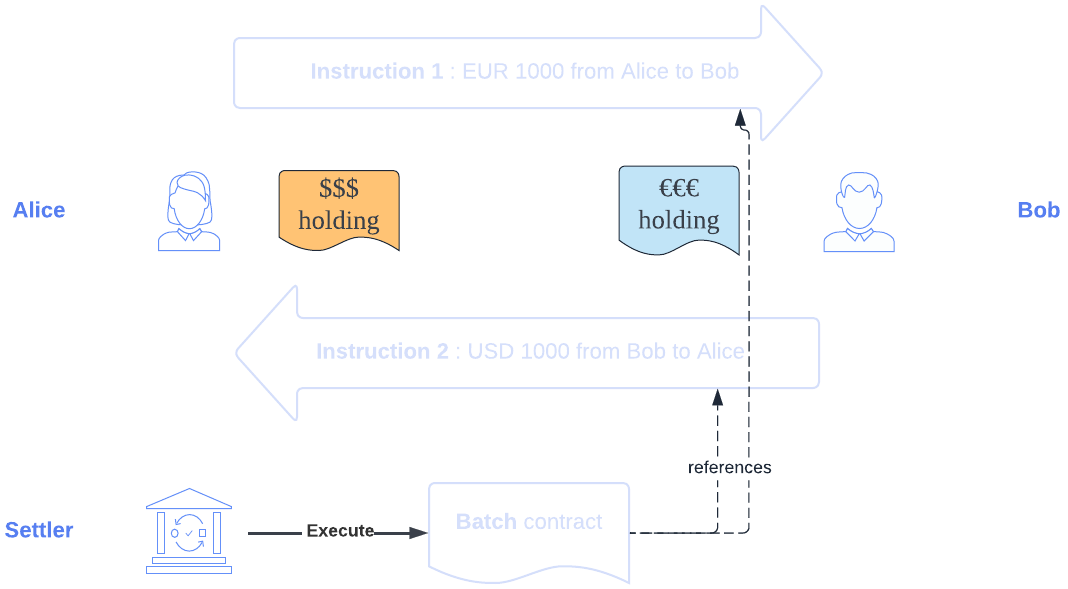


Each receiver can then specify to which account the holding should be sent by *approving* the corresponding instruction.



Execute

Once both instructions are allocated and approved, a Settler party uses the *Batch* contract to *execute* them and finalize settlement in one atomic transaction.



The instructions and the batch are archived following a successful execution.

Remarks

There are some assumptions that need to hold in order for the settlement to work in practice:

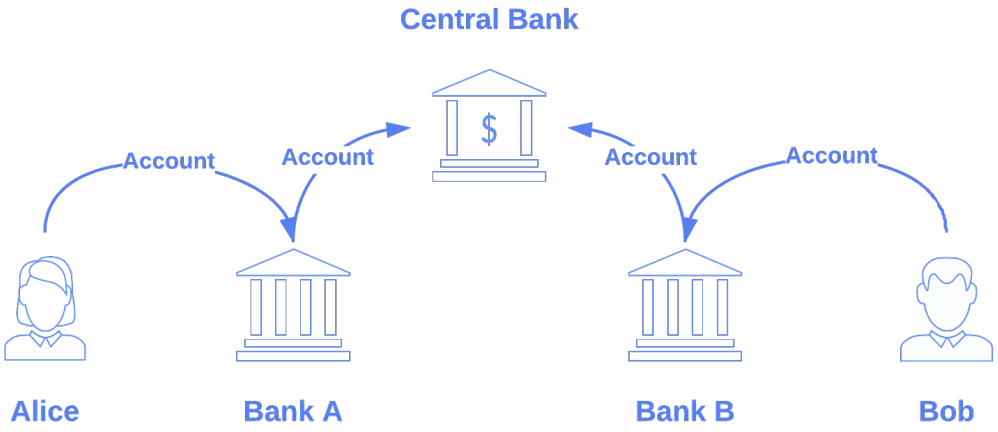
> Bob needs to have an account at the custodian where Alice's holding is held and vice versa (for an example with intermediaries, see *Route provider* below.
> Both holdings need to be *Transferable*
> The transfer must be fully authorized (i.e., the parties allocating and approving an instruction must be the controllers of outgoing and incoming transfers of the corresponding accounts, respectively)

Also, note that the allocation and approval steps can happen in any order.

1.18.2.2 The components in detail

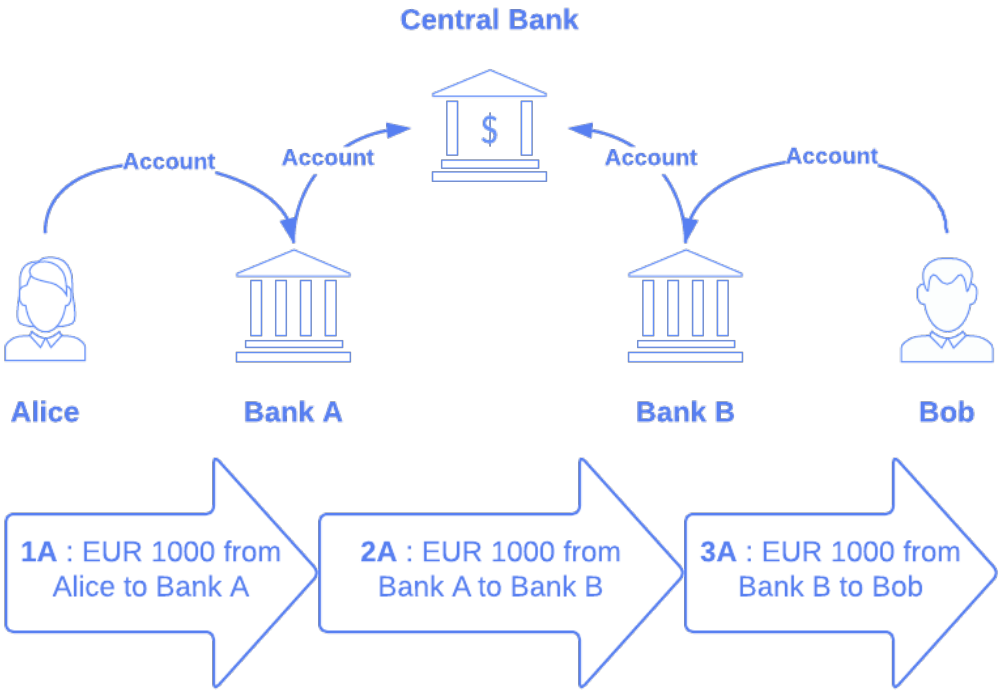
Route provider

When a transfer requires intermediaries to be involved, the role of a *Route Provider* becomes important. Let us assume, for instance, that Alice's EUR holding in the example above is held at Bank A, whereas Bob has a EUR account at Bank B. Bank A and Bank B both have accounts at the Central Bank.



In this case, a direct holding transfer from Alice to Bob cannot generally be instructed. The original *Instruction* between Alice and Bob needs to be replaced by three separate *Instructions*:

1A: Alice sends EUR 1000 (held at Bank A) to Bank A
1B: Bank A sends EUR 1000 (held at the Central Bank) to Bank B.
1C: Bank B credits EUR 1000 to Bob's account (held at Bank B)



We refer to this scenario as *settlement with intermediaries*, or just *intermediated settlement*.

The Route Provider is used to discover a settlement route, i.e., *routed steps*, for each settlement *step*.

Settlement factory

The *Settlement Factory* is used to instruct settlement, i.e., create the *Batch* contract and the settlement *Instructions*, from *routed steps*, so that they can be allocated and approved by the respective parties.

Instruction

The *Instruction* is used to settle a single holding transfer at a specific custodian, once it is `allocated` and `approved`.

In the *Allocation* step, the sender acknowledges the transfer and determines how to send the holding. This is usually done by allocating with a *Pledge* of the sender's existing holding (which has the correct instrument quantity) at the custodian. When the sender is also the custodian, the instruction can be allocated with *CreditReceiver*. In this case, a new holding is directly credited into the receiver's account.

In the *Approval* step, the receiver acknowledges the transfer and determines how to receive the holding. This is usually done by approving with *TakeDelivery* to one of the receiver's accounts at the custodian. When the receiver is also the incoming holding's custodian, the instruction can be approved with *DebitSender*. In this case, the holding is directly debited from the sender's account. A holding owned by the custodian at the custodian has no economical value, it is a liability against themselves and can therefore be archived without consequence.

To clarify these concepts, here is how the 3 instructions in the intermediated example above would be allocated / approved.

Instruction	Allocation	Approval
1A : EUR 1000 from Alice to Bank A @ Bank A	Alice pledges her holding	Bank A approves with DebitSender
1B : EUR 1000 from Bank A to Bank B @ Central Bank	Bank A pledges its holding	Bank B takes delivery to its account
1C : EUR 1000 from Bank B to Bob @ Bank B	Bank B allocates with CreditReceiver	Bob takes delivery to his account

Finally, the *Instruction* supports two additional settlement modes:

> Any instruction can settle off-ledger (if the stakeholders agree to do so). For this to work, we require the custodian and the sender to jointly allocate the instruction with a *SettleOffledger*, and the custodian and the receiver to jointly approve the instruction with a *SettleOffledgerAcknowledge*.
>
> A special case occurs when a transfer happens via an intermediary at the same custodian, i.e., we have 2 instructions having the same custodian and instrument quantity (in a batch), and the receiver of the first instruction is the same as the sender of the second instruction. In this case, we allow the holding received from the first instruction to be passed through to settle the second instruction, i.e., without using any pre-existing holding of the intermediary. For this to work, the first instruction is approved with *PassThroughTo* (i.e., pass through to the second instruction), and the second instruction is allocated with *PassThroughFrom* (i.e., pass through from the first instruction). An intermediary account used for the passthrough is thereby also to be specified.

Batch

The *Batch* is used to execute a set of instructions atomically. Execution will fail if any of the *Instructions* is not fully allocated / approved, or if the transfer is unsuccessful.

1.18.2.3 Remarks and further references

The settlement concepts are also explored in the *Settlement tutorial*.

1.18.3 Lifecycling

Lifecycling refers to the evolution of financial instruments over their lifetime. This includes processing of contractual events, like interest payments or coupon cashflows, as well as discretionary events, like dividends and other corporate actions. The library provides a standard mechanism for processing such events across different instruments.

The interfaces for lifecycling are defined in the `Daml.Finance.Interface.Lifecycle` package, and several default implementations are provided in the `Daml.Finance.Lifecycle` package.

In this section, we first motivate the particular approach to lifecycling in Daml Finance. We then explain the process in detail using the example of a cash dividend event. Finally, we describe each of the involved components in depth.

1.18.3.1 Approach

Single Source of Truth

A general principle we follow in Daml Finance is that there should only be a single instance of any given instrument on the ledger. This instance is centrally maintained by the issuer of the instrument, with the possibility for a depository to act as an additional, third-party trust anchor. As part of lifecycling, this single instance of the instrument produces lifecycle effects, which are then used across all parties on the ledger as a single source of truth for how to process a certain event. This avoids any duplication of lifecycling logic or redundant processing of instruments, and therefore removes the need for reconciliation across parties involved in the process.

Instrument Versioning

In current financial markets instruments are usually referenced by a textual identifier, like an ISIN or CUSIP number. When a particular corporate action is processed on its effective date the instrument referred to by an identifier changes implicitly. As an example, the ISIN for a stock refers to the cum-dividend instrument (where holders are still entitled to the dividend) up until the ex-dividend date. From the ex-dividend date onwards the same ISIN refers to the ex-dividend instrument, so any stock acquired on or after that date is not entitled to the dividend anymore. This leads to a lot of complexity during processing of such corporate actions. In particular, it forces these events to be processed in a big bang approach, as a consistent snapshot of holdings needs to be taken to determine the rightful recipients of any resulting cashflows.

In Daml Finance we aim for a more efficient and flexible operating model for processing lifecycle events. All instruments are strictly versioned so that we can clearly differentiate between the cum-

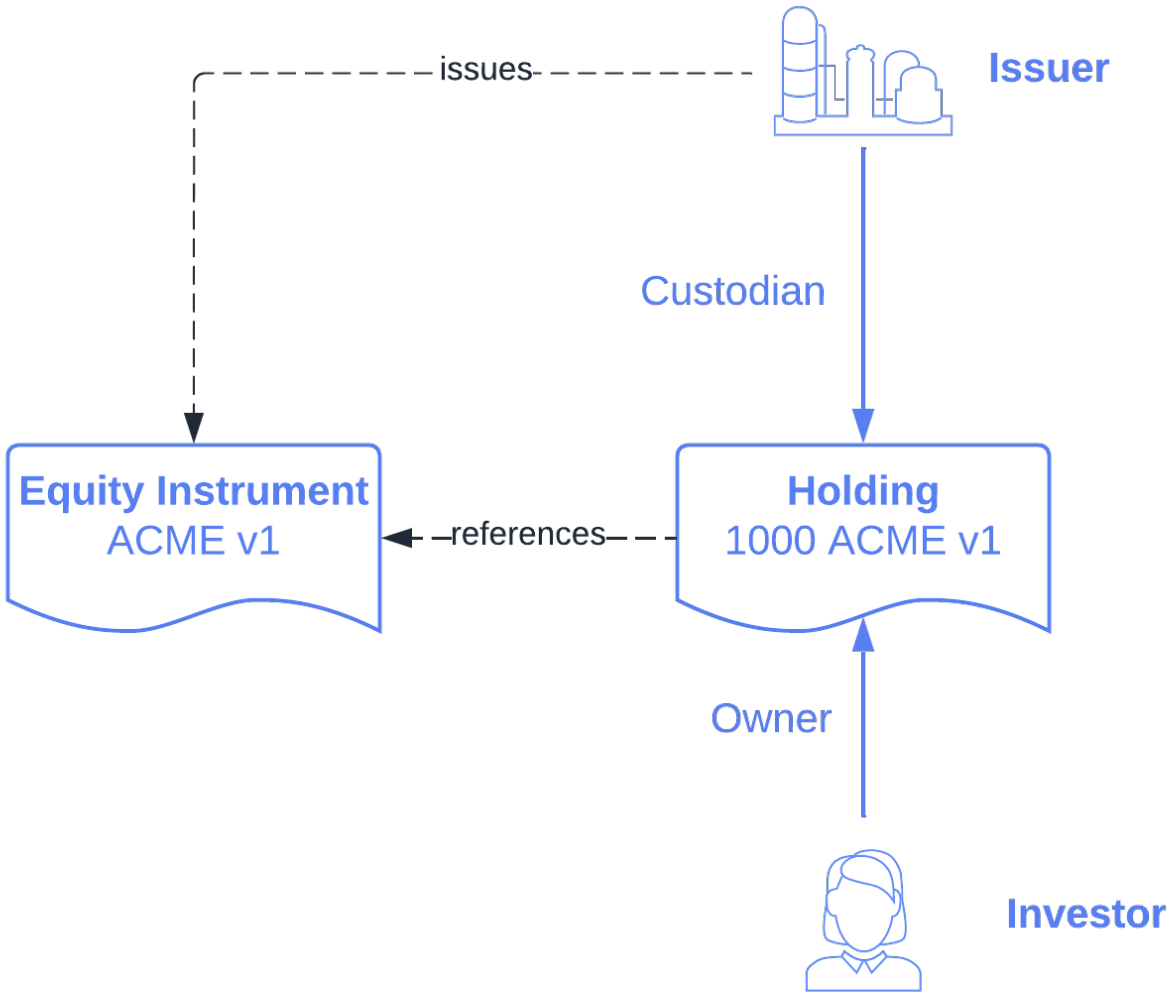
and ex-event version of an instrument. This means that it is perfectly safe for those versions to co-exist at the same time, and it allows for a gradual transition from one version to another. Generally, the issuer of an instrument is responsible for creating and maintaining instrument versions, and for producing the cashflow effects of a particular lifecycle event. During the lifecycle process, holders of this instrument will migrate their holdings to a new version of the instrument while at the same time claiming any resulting cashflows from the event.

Versions are usually considered opaque strings, but one can follow a numerical versioning scheme if an instrument is known to have linear evolution (i.e., there is no optionality that can result in two different evolution paths).

1.18.3.2 Workflow

In this scenario we go through the process of paying a cash dividend from an issuer to an investor. The initial state looks as follows:

An issuer maintains an `ACME` instrument representing shares in a company
An investor owns a holding of 1000 units of the `ACME` instrument (version 1) with the issuer
The issuer wants to process and pay a cash dividend of USD 10.0 per unit of its `ACME` instrument (version 1)

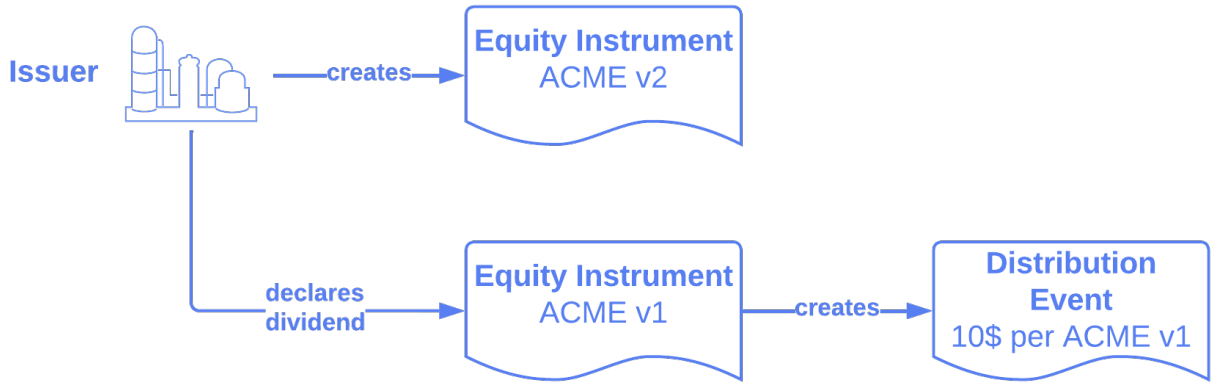


We will now explain each step in the process in detail.

Creating the event

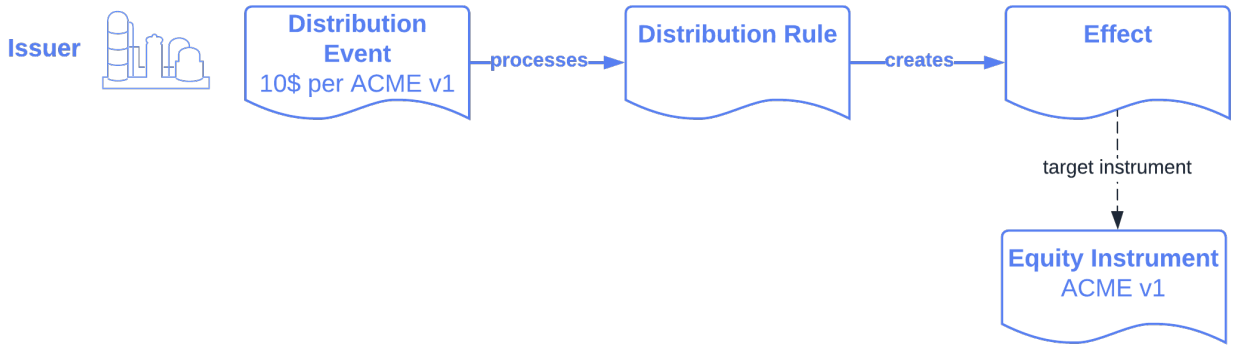
The issuer first creates a new instance of the `ACME` instrument, assigning a new version. Note that the logic to create the new version of an instrument can also be encoded in the lifecycle rule. The new version is then automatically produced when processing the event as described in the next step.

Now, the issuer creates a lifecycle event defining the terms of dividend. In our example we can use the `DistributeDividend` choice on the *Equity* instrument to create such an event. This is merely a convenience choice available for equities, any workflow can be used to create new instrument versions and associated lifecycle events.



Processing the event

The event is now passed into a *Distribution Rule*, which generates the *Lifecycle Effect* describing the distribution of assets per unit of `ACME` stock. The effect refers to a `targetInstrument`, which is the version of the instrument that can be used by stock holders to claim the cash dividend according to the number of stocks held. By being tied to a specific version we prevent holders from (accidentally or intentionally) claiming a particular effect twice.

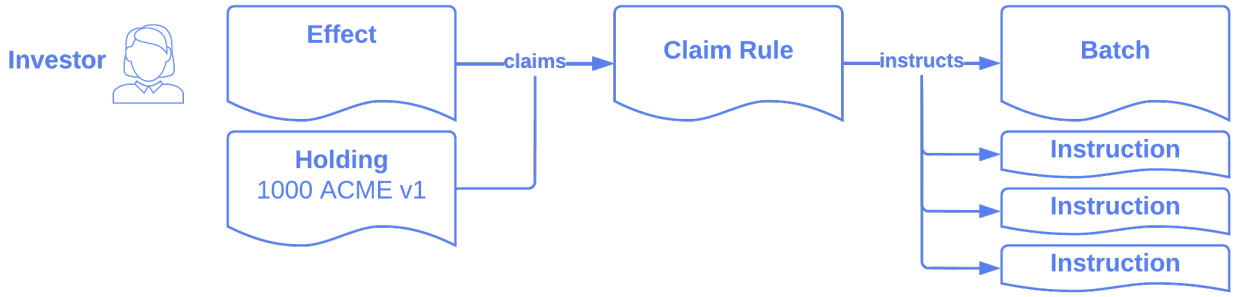


Claiming the effect

The investor can now present its holding of `ACME` stock along with the corresponding *Effect* to a *Claim Rule*. This will instruct settlement for:

The exchange of `ACME` stock versions held: the investor sends back the old version, and receives the new one
The payment of the cash dividend amount corresponding to the number of stocks held

Both legs of this settlement are grouped in a *Batch* to provide atomicity. This ensures that the investor can never claim a dividend twice, as after settlement they only hold the new version of the stock, which is not entitled to the dividend anymore.



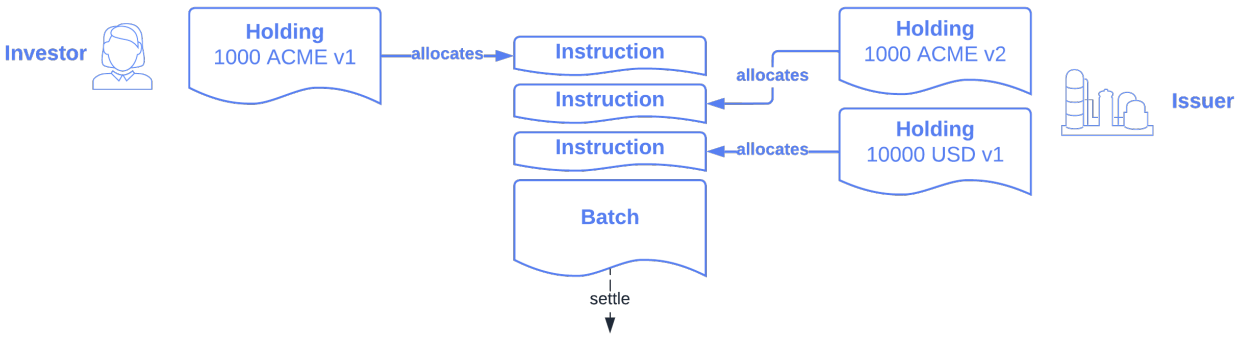
Note that the party responsible for claiming an effect can be specified flexibly in the *Claim Rule* contract. Through this contract, custodians can be given the authority to push a given corporate action to the asset holder as is common in current operating procedures.

The model also supports atomic, intermediated settlement of lifecycle events. For example, if a dividend is to be settled between issuer, custodian, and investor in a single transaction, the custodian (having visibility of both its holding at the issuer and the investor's holding) can pass in both holdings into the claim rule, and thereby instruct a single batch to settle both sides.

Settlement

The batch and instructions resulting from claiming an effect can now be settled as described in the *Settlement* section of the documentation.

The following picture shows the three asset movements involved in this particular example:



The result of processing the settlement batch results in the investor receiving a 10000 USD dividend and 1000 shares of `ACME` v2 in return for their 1000 shares of `ACME` v1.

1.18.3.3 Components

Events

The *Event* interface describes basic properties of a lifecycle event:

The event providers
The event identifier and description
The event timestamp

Different implementations exist to cover typical event types:

The *Distribution* event can be used to distribute assets to holders of an instrument. This covers cash-, share-, and mixed dividends, rights issues, or the distribution of voting rights.
The *Replacement* event handles replacements of one instrument for another with support for a factor. This covers corporate actions like (reverse) stock splits, mergers, and spin-offs.

Lifecycle Rule

The *Lifecycle Rule* is used to process an event and calculate the resulting lifecycle effect. A lifecycle rule can either assume that a new version of the instrument has already been created (as is the case for the *Distribution* and *Replacement* rules), or it can create the new version of the instrument as part of its implementation. The latter can be useful if information required to create the new version is only available upon processing of the event, as is the case for *Generic Instrument* evolution, as well as other *Contingent Claims* based instruments.

Lifecycling of Contingent Claims based instruments can be divided into two categories:

Time based evolution: An instrument is evolved solely due to the passage of time. An example is a fixed coupon bond, where a coupon payment is due at the end of every coupon period. This can be *automatically* lifecycled by providing the event time (and any observables required). The tutorial *Time-based lifecycling (using a fixed rate bond)* describes how this is done.
Election based evolution: An instrument is evolved as a result of a *manual* election. One example is a callable bond, where the custodian of the corresponding holding has the right (but not the obligation) to call, or redeem early, the instrument on certain call dates. Lifecycling of such an instrument requires an *Election* event. Time alone is not sufficient, because the evolution of the instrument depends on manual actions of the holding stakeholders. Check out the tutorial *Election-based lifecycling (using a callable bond)* for more details on how this can be implemented in practice.

Note that some instruments can require both types of lifecycling. An example of this is a callable bond that is callable only on some of the coupon dates.

Claim Rule

The *Claim Rule* is used to claim lifecycle effects and instruct settlement thereof. Each effect specifies a target instrument (and version), and holdings on this instrument (version) are required to claim an effect. This serves as proof of ownership such that there is no need for an issuer to take a consistent snapshot of holdings as of a specific date.

The output of the claim rule is a *Batch* and a set of *Instruction* s forming an atomic unit of settlement.

Note that multiple holdings can be passed into the claim rule in order to instruct intermediated settlement of an effect, or to instruct atomic settlement for multiple asset holders at the same time.

Effects

An *Effect* describes the asset movements resulting from a particular event. It specifies these movements per unit of a target instrument and version. Holdings on this specific instrument version entitle a holder to claim the effect, which results in the required asset movements to be instructed.

1.19 Instruments

This section describes which instruments are included out of the box in Daml Finance. Each instrument package contains a list of supported instruments. The instrument extension pages explain what each instrument does and how to set it up.

1.19.1 Bonds

The following instruments are included in the *Bond Instrument package*:

Fixed rate bonds
Floating rate bonds
Callable bonds
Inflation linked bonds
Zero coupon bonds

1.19.2 Equites

The following instruments are included in the *Equity Instrument package*:

Equities (can also be used to model ETFs)

1.19.3 Options

The following instruments are included in the *Option Instrument package*:

Cash-settled European options
Physically-settled European options
Barrier options
Dividend options

1.19.4 Swaps

The following instruments are included in the *Swap Instrument package*:

Interest rate swaps
Currency swaps
Foreign exchange swaps
Credit default swaps
Asset swaps
FpML swaps (supports the above swap types using the FpML schema)

1.19.5 Other Instruments

In addition to the above instruments, which model specific payoffs, the library provides

a *Token Instrument*, whose terms are defined by a simple textual label
a *Generic Instrument*, which provides a flexible framework to structure user-defined payoffs and lifecycle them on the ledger

1.19.6 How to use the Token Instrument packages

A Token is a simple instrument template whose economic terms on the ledger are defined by two textual fields, namely an `id` and a `description`.

It is often used to model financial instruments that do not exhibit complex lifecycling logic, such as currencies.

1.19.6.1 How to create a Token Instrument

The following code snippets are taken from the *Getting Started tutorial*, which you can install using the Daml assistant.

In order to instantiate a Token Instrument, we first need to create the corresponding instrument factory template

```
tokenFactoryCid <- toInterfaceContractId @Token.F <$> submit bank do
  createCmd Token.Factory with
    provider = bank
    observers = empty
```

We can then specify the terms of the instrument and exercise the `Create` choice in the factory to create the token.

```
let
  instrumentId = Id "USD"
  instrumentVersion = "0"
  instrumentKey = InstrumentKey with
    issuer = bank
    depository = bank
    id = instrumentId
    version = instrumentVersion
now <- getTime

submit bank do
  exerciseCmd tokenFactoryCid Token.Create with
    token = Token with
      instrument = instrumentKey
      description = "Instrument representing units of a generic token"
      validAsOf = now
    observers = empty
```

1.19.6.2 How to lifecycle a Token Instrument

Generic corporate actions, such as *Distribution events*, can be applied to Token Instruments. The *Lifecycling* section of the Getting Started tutorial shows how this is done in detail.

1.19.7 How to use the Bond Instrument packages

To follow the code snippets used in this page in Daml Studio, you can clone the Daml Finance repository and run the scripts included in the Instrument/Bond/Test folder.

1.19.7.1 How to use a Bond Instrument in your application

As explained in the *Getting Started* section and on the *Architecture* page, your app should only depend on the interface layer of Daml Finance. For bonds this means that you should only include the *bond interface package*.

Your initialization scripts are an exception, since they are only run once when your app is initialized. These are used to create the necessary instrument factories. Your app can then create bond instruments through these factory contracts.

1.19.7.2 How to Create a Bond Instrument

There are different types of bonds, which mainly differ in the way the coupon is defined. In order to create a bond instrument you first have to decide what type of bond you need. The *bond instrument package* currently supports the following bond types:

Fixed Rate

Fixed rate bonds pay a constant coupon rate at the end of each coupon period. The coupon is quoted on a yearly basis (per annum, p.a.), but it could be paid more frequently (e.g. quarterly). For example, a bond could have a 2% p.a. coupon and a 6M coupon period. That would mean a 1% coupon is paid twice a year.

As an example we will create a bond instrument paying a 1.1% p.a. coupon with a 12M coupon period. This example is taken from Instrument/Bond/Test/FixedRate.daml , where all the details are available.

We start by defining the terms:

```
let
  issueDate = date 2019 Jan 16
  firstCouponDate = date 2019 May 15
  maturityDate = date 2020 May 15
  notional = 1.0
  couponRate = 0.011
  couponPeriod = M
  couponPeriodMultiplier = 12
  dayCountConvention = Act365Fixed
  businessDayConvention = Following
```

The *day count convention* is used to determine how many days, i.e., what fraction of a full year, each coupon period has. This will determine the exact coupon amount that will be paid each period.

The *business day convention* determines *how* a coupon date is adjusted if it falls on a non-business day.

We also need holiday calendars, which determine *when* to adjust dates.

We can use these variables to create a *PeriodicSchedule*:

```
  let
    (y, m, d) = toGregorian firstCouponDate
    periodicSchedule = PeriodicSchedule with
      businessDayAdjustment =
        BusinessDayAdjustment with
          calendarIds = holidayCalendarIds
          convention = businessDayConvention
      effectiveDateBusinessDayAdjustment = None
      terminationDateBusinessDayAdjustment = None
      frequency =
        Periodic Frequency with
          rollConvention = DOM d
          period = Period with
            period = couponPeriod
            periodMultiplier = couponPeriodMultiplier
      effectiveDate = issueDate
      firstRegularPeriodStartDate = Some firstCouponDate
      lastRegularPeriodEndDate = Some maturityDate
      stubPeriodType = None
      terminationDate = maturityDate
```

This is used to determine the periods that are used to calculate the coupon. There are a few things to note here:

The *RollConventionEnum* defines whether dates are rolled at the end of the month or on a given date of the month. In our example above, we went for the latter option.
The *StubPeriodTypeEnum* allows you to explicitly specify what kind of stub period the bond should have. This is optional and not used in the example above. Instead, we defined the stub implicitly by specifying a `firstRegularPeriodStartDate`: since the time between the issue date and the first regular period start date is less than 12M (our regular coupon period), this implies a short initial stub period.

Now that we have defined the terms we can create the bond instrument:

```
    let
      instrument = InstrumentKey with
        issuer
        depository
        id = Id label
        version = "0"

    cid <- submitMulti [issuer] [publicParty] do
      exerciseCmd fixedRateBondFactoryCid FixedRate.Create with
        fixedRate = FixedRate with
          instrument
          description
          couponRate
          periodicSchedule
          holidayCalendarIds
          calendarDataProvider
          dayCountConvention
          currency
          notional
          lastEventTimestamp
        observers = M.fromList observers
```

Once this is done, you can create a holding on it using *Account.Credit*.

Floating Rate

Floating rate bonds pay a coupon which is determined by a reference rate. There is also a rate spread, which is paid in addition to the reference rate.

Here is an example of a bond paying Euribor 3M + 1.1% p.a. with a 3M coupon period:

```
  let
    issueDate = date 2019 Jan 16
    firstCouponDate = date 2019 Feb 15
    maturityDate = date 2019 May 15
    referenceRateId = "EUR/EURIBOR/3M"
    notional = 1.0
    couponSpread = 0.011
    couponPeriod = M
    couponPeriodMultiplier = 3
    dayCountConvention = Act365Fixed
    businessDayConvention = Following
```

Using these terms we can create the floating rate bond instrument:

```
  let
    instrument = InstrumentKey with
      issuer
      depository
      id = Id label
      version = "0"

  cid <- submitMulti [issuer] [publicParty] do
    exerciseCmd floatingRateBondFactoryCid FloatingRate.Create with
      floatingRate = FloatingRate.FloatingRate with
        instrument
        description
        referenceRateId
        couponSpread
        periodicSchedule
        holidayCalendarIds
        calendarDataProvider
        dayCountConvention
        currency
        notional
        lastEventTimestamp
      observers = M.fromList observers
```

The reference rate (Euribor 3M) is observed once at the beginning of each coupon period and used for the coupon payment at the end of that period.

Callable

Callable bonds are similar to the bonds above, but in addition they can be redeemed by the issuer before maturity. The callability is restricted to some (or all) of the coupon dates. In other words, these bonds have a *Bermudan* style embedded call option.

Both fixed and floating rate coupons are supported by this instrument. In case of a floating rate, there is often a fixed spread as well. This can be represented by a fixed rate coupon, which is shown in the following example. Here is a bond paying Libor 3M + 0.1% p.a. with a 3M coupon period:

```
-- Libor + 0.1% coupon every 3M (with a 0% floor and a 1.5% cap)
let
  rollDay = 15
  issueDate = date 2022 Jan rollDay
  firstCouponDate = date 2022 Apr rollDay
  maturityDate = date 2024 Jan rollDay
  notional = 1.0
  couponRate = 0.001
  capRate = Some 0.015
  floorRate = Some $ 0.0
  couponPeriod = M
  couponPeriodMultiplier = 3
  dayCountConvention = Act360
  useAdjustedDatesForDcf = True
  businessDayConvention = Following
  referenceRateId = "USD/LIBOR/3M"
  floatingRate = Some FloatingRate with
    referenceRateId
```

(continues on next page)

(continued from previous page)

```
      referenceRateType = SingleFixing CalculationPeriodStartDate
      fixingDates = FixingDates with
        periodMultiplier = -2
        period = D
        dayType = Some Business
        businessDayConvention = NoAdjustment
        businessCenters = ["USD"]
```

The coupon rate in this example also has a 0% floor and a 1.5% cap. This is configurable, just set the cap or floor to *None* if it does not apply.

The fixed rate is fairly simple to define, but the floating rate requires more inputs. A *FloatingRate* data type is used to specify which reference rate should be used and on which date the reference rate is fixed for each coupon period.

The above variables can be used to create a *couponSchedule*:

```
    couponSchedule = createPaymentPeriodicSchedule firstCouponDate
↪holidayCalendarIds
      businessDayConvention couponPeriod couponPeriodMultiplier issueDate
↪maturityDate
```

This *couponSchedule* is used to determine the coupon payment dates, where the *businessDayConvention* specifies how dates are adjusted. Also, *useAdjustedDatesForDcf* determines whether adjusted or unadjusted dates should be used for day count fractions (to determine the coupon amount).

In addition to the Libor/Euribor style reference rates, compounded SOFR and similar reference rates are also supported. In order to optimize performance, these compounded rates are calculated via a (pre-computed) continuously compounded index, as described in the *ReferenceRateTypeEnum*. For example, here is how *daily compounded SOFR* can be specified using the *SOFR Index*:

```
    referenceRateId = "SOFR/INDEX"
    floatingRate = Some FloatingRate with
      referenceRateId
      referenceRateType = CompoundedIndex Act360
```

This instrument also allows you to configure on which coupon dates the bond is callable. This is done by specifying a separate *callSchedule*. The bond is callable on the *last* date of each schedule period. For example, if the bond is callable on every coupon date, simply set *callSchedule = couponSchedule*. Alternatively, if the bond is only callable every six months, this can be configured by specifying a different schedule:

```
    -- Define a schedule for callability. The bond is callable on the *last* date
↪of each schedule
    -- period.
    -- In this example, it is possible to call the bond every 6M (every second
↪coupon date).
    callScheduleStartDate = issueDate
    callScheduleEndDate = maturityDate
    callPeriod = couponPeriod
    callPeriodMultiplier = 6
    callScheduleFirstRegular = None -- Only used in case of an irregular schedule
    callSchedule = createPeriodicSchedule callScheduleFirstRegular
↪holidayCalendarIds
      businessDayConvention callPeriod callPeriodMultiplier callScheduleStartDate
```

(continues on next page)

(continued from previous page)

```
    callScheduleEndDate rollDay
  noticeDays = 5
```

The *noticeDays* field defines how many business days notice is required to call the bond. The election whether or not to call the bond must be done on this date.

Using these terms we can create the callable bond instrument:

```
  let
    instrument = InstrumentKey with
      issuer
      depository
      id = Id label
      version = "0"

  cid <- submitMulti [issuer] [publicParty] do
    exerciseCmd callableBondFactoryCid Callable.Create with
      callable = Callable with
        instrument
        description
        floatingRate
        couponRate
        capRate
        floorRate
        couponSchedule
        noticeDays
        callSchedule
        holidayCalendarIds
        calendarDataProvider
        dayCountConvention
        useAdjustedDatesForDcf
        currency
        notional
        lastEventTimestamp
        prevElections = []
      observers = M.fromList observers
```

Unlike regular fixed and floating bonds, which are lifecycled based on the passage of time, this callable bond instrument contains an embedded option that is not automatically exercised. Instead, the custodian of the bond holding must manually decide whether or not to call the bond. This is done by making an *Election*.

This callable bond example is taken from Instrument/Bond/Test/Callable.daml , where all the details are available. Also, check out the *Election based lifecycling tutorial* for more details on how to define and process an *Election* in practice. Note that the sample bond above, which is callable only on some of the coupon dates, will require two types of lifecycling:

> *Time based lifecycling* on coupon dates when the bond is *not* callable.
> *Election based lifecycling* on coupon dates when the bond *is* callable.

Inflation Linked

Inflation linked bonds pay a fixed coupon rate at the end of every coupon period. The coupon is calculated based on a principal that is adjusted according to an inflation index, for example the Consumer Price Index (CPI) in the U.S.

Here is an example of a bond paying 1.1% p.a. (on a CPI adjusted principal) with a 3M coupon period:

```
let
  issueDate = date 2019 Jan 16
  firstCouponDate = date 2019 Feb 15
  maturityDate = date 2019 May 15
  inflationIndexId = "CPI"
  notional = 1.0
  couponRate = 0.011
  couponPeriod = M
  couponPeriodMultiplier = 3
  dayCountConvention = Act365Fixed
  businessDayConvention = Following
```

Based on these terms we can create the inflation linked bond instrument:

```
  let
    instrument = InstrumentKey with
      issuer
      depository
      id = Id label
      version = "0"

  cid <- submitMulti [issuer] [publicParty] do
    exerciseCmd inflationLinkedBondFactoryCid InflationLinked.Create with
      inflationLinked = InflationLinked with
        instrument
        description
        periodicSchedule
        holidayCalendarIds
        calendarDataProvider
        dayCountConvention
        couponRate
        inflationIndexId
        inflationIndexBaseValue
        currency
        notional
        lastEventTimestamp
      observers = M.fromList observers
```

At maturity, the greater of the adjusted principal and the original principal is redeemed. For clarity, this only applies to the redemption amount. The coupons are always calculated based on the adjusted principal. This means that in the case of deflation, the coupons would be lower than the specified coupon rate but the original principal would still be redeemed at maturity.

Zero Coupon

A *zero coupon bond* does not pay any coupons at all. It only pays the redemption amount at maturity.

Here is an example of a zero coupon bond:

```
let
  issueDate = date 2019 Jan 16
  maturityDate = date 2020 May 15
  notional = 1000.0
```

Based on this we create the zero coupon bond instrument:

```
  let
    instrument = InstrumentKey with
      issuer
      depository
      id = Id label
      version = "0"

  cid <- submitMulti [issuer] [publicParty] do
    exerciseCmd zeroCouponBondFactoryCid ZeroCoupon.Create with
      zeroCoupon = ZeroCoupon with
        instrument
        description
        currency
        issueDate
        maturityDate
        notional
        lastEventTimestamp
      observers = M.fromList observers
```

1.19.7.3 Frequently Asked Questions

How do I transfer or trade a bond?

When you have created a holding on a bond instrument this can be transferred to another party. This is described in the *Getting Started: Transfer* tutorial.

In order to trade a bond (transfer it in exchange for cash) you can also initiate a delivery versus payment with atomic settlement. This is described in the *Getting Started: Settlement* tutorial.

How do I process coupon payments for a bond?

On the coupon payment date, the issuer will need to lifecycle the bond. This will result in a lifecycle effect for the coupon, which can be cash settled. This is described in detail in the *Lifecycling* and the *Intermediated Lifecycling* tutorials.

How do I redeem a bond?

On the redemption date, both the last coupon and the redemption amount will be paid. This is processed in the same way as a single coupon payment described above.

How do I view the terms of a bond instrument?

There are several ways to access the data of a contract, as explained in the *Patterns* section.

1.19.8 How to use the Equity Instrument packages

To follow the code snippets used in this page in Daml Studio, you can clone the Daml Finance repository and run the scripts included in the Instrument/Equity/Test/ folder.

1.19.8.1 How to use an Equity Instrument in your application

As explained in the *Getting Started* section and on the *Architecture* page, your app should only depend on the interface layer of Daml Finance. For equities this means that you should only include the *equity interface package*.

Your initialization scripts are an exception, since they are only run once when your app is initialized. These are used to create the necessary instrument factories. Your app can then create equity instruments through these factory contracts.

1.19.8.2 The Equity Interface

The equity instrument supports different lifecycle events, such as dividends, stock splits and mergers. These are modeled using the choices on the *Equity interface*, namely `DeclareDistribution`, `DeclareReplacement` and `DeclareStockSplit`. We will now demonstrate each one with a concrete lifecycle event.

1.19.8.3 Dividend

The most common lifecycle event of an equity is probably dividends. This normally means that the holder of a stock receives a given amount of cash for each stock held. This is modeled using the `DeclareDistribution` choice. It creates a *Distribution Event*, which allows you to specify distribution per share. In the case of a cash dividend, this would be a cash instrument. However, the company can also choose to distribute additional stock or even stock options. Since the *Distribution Event* supports an arbitrary `perUnitDistribution` instrument, it can be used to model those use cases as well.

In order to process a lifecycle event, you have to create two versions of the instrument: one before the event and one after the event. In the case of a dividend event, this means one instrument *cum* dividend (which includes the dividend) and one *ex* dividend (which does no longer include the dividend):

```
  cumEquityInstrument <- originateEquity issuer issuer "EQUITY-INST-1" "0" "ABC"
↪pp now
  exEquityInstrument <- originateEquity issuer issuer "EQUITY-INST-1" "1" "ABC"
↪[] now
```

Once this is done, you can create a holding on it. This is not limited to integer holdings, but fractional holdings are supported as well:

```
  -- Distribute holdings: fractional holdings are also supported.
  investorEquityCid <- Account.credit [publicParty] cumEquityInstrument 1000.25
↪investorAccount
```

We create a distribution rule for the cash dividend. It defines the business logic for the dividend and it has the issuer as signatory:

```
  -- Create cash dividend rule
  distributionRuleCid <- toInterfaceContractId @Lifecycle.I <$> submit issuer do
    createCmd Distribution.Rule with
      providers = S.singleton issuer
      lifecycler = issuer
      observers = S.singleton publicParty
      id = Id "LifecycleRule"
      description = "Rule to lifecycle an instrument following a distribution
↪event"
```

We also need a distribution event, which defines the terms of the dividend. In this case, it is USD 2 cash per share (this also works for a fractional amount of shares):

```
  -- Create cash dividend event: USD 2 per share (this also works with fractional
↪shares)
  distributionEventCid <-
    Instrument.submitExerciseInterfaceByKeyCmd @Equity.I [issuer] []
↪cumEquityInstrument
      Equity.DeclareDistribution with
        id = Id $ "ABC - " <> show now
        description = "Cash Dividend"
        effectiveTime = now
        newInstrument = exEquityInstrument
        perUnitDistribution = [qty 2.0 cashInstrument]
```

This allows the issuer to lifecycle the instrument by exercising the `Evolve` choice:

```
  -- Lifecycle cash dividend
  (_, [effectCid]) <- submit issuer do
    exerciseCmd distributionRuleCid Lifecycle.Evolve with
      observableCids = []
      eventCid = distributionEventCid
      instrument = cumEquityInstrument
```

This results in a lifecycle effect, which can be settled. The settlement of effects is covered in the *Lifecycling tutorial*.

1.19.8.4 Bonus issue

Instead of a cash dividend, a company may also decide to offer free shares (or warrants) instead of cash to current shareholders. This is called *bonus issue* and it is modeled in a similar way to the *dividend* above. The main difference is in the distribution event, which now distributes a different instrument (equity instead of cash):

```
  -- Create bonus issue event: receive 2 additional shares for each share
↪currently held
  -- (this also works with fractional shares)
  distributionEventCid <-
    Instrument.submitExerciseInterfaceByKeyCmd @Equity.I [issuer] []
↪cumEquityInstrument
      Equity.DeclareDistribution with
        id = Id $ "ABC - " <> show now
        description = "Bonus issue"
        effectiveTime = now
        newInstrument = exEquityInstrument
        perUnitDistribution = [qty 2.0 exEquityInstrument]
```

Similarly, if there is a bonus issue that awards warrants instead of equity, that can be modeled in the same way. Just replace the equity instrument by a warrant instrument on the `perUnitDistribution` line above.

1.19.8.5 Dividend option

A company may give shareholders the option of choosing what kind of dividend they want to receive. For example, a shareholder could choose between a dividend in cash *or* in stock. Currently, there are two different ways this can be modeled in the library:

1. Using a dividend option instrument

The preferred way is to model this using the following two components:

> A dividend option instrument, which describes the economic terms of the rights a shareholder receives. The page on the *Option Instrument package* describes how to create a physically settled *Dividend* option.
> The `DeclareDistribution` choice to distribute the above option instrument in the correct proportion (e.g. 1 option contract for each share held). This can be done in the same way as the *Bonus Issue* example described earlier, just change the `perUnitDistribution` line to distribute the option instrument you created above.

When current shareholders receive the option instrument they can choose between one of the dividend payment types offered by the issuer, for example cash in a foreign currency.

More details on this dividend option process are described in Instrument/Equity/Test/DivOption.daml , in particular how to define and process an *Election*.

2. Using multiple distribution events

The `DeclareDistribution` choice can be used for this as well. The issuer creates one event for each dividend option that shareholders can choose from:

```
  -- Create dividend option event.
  -- For each share currently held, the shareholder can choose to either receive
↪cash (USD 10.5) or
  -- stock (1.5 additional shares).
  -- perUnitDistribution is an arbitrary list, so this can be extended with
↪additional options, e.g.
  -- warrants or cash in a different currency.
  distributionEventCashCid <-
    Instrument.submitExerciseInterfaceByKeyCmd @Equity.I [issuer] []
↪cumEquityInstrument
      Equity.DeclareDistribution with
        id = Id $ "ABC - " <> show now
        description = "Dividend option: cash"
        effectiveTime = now
        newInstrument = exEquityInstrument
        perUnitDistribution = [qty 10.5 cashInstrument]

  distributionEventStockCid <-
    Instrument.submitExerciseInterfaceByKeyCmd @Equity.I [issuer] []
↪cumEquityInstrument
      Equity.DeclareDistribution with
        id = Id $ "ABC - " <> show now
        description = "Dividend option: stock"
        effectiveTime = now
        newInstrument = exEquityInstrument
        perUnitDistribution = [qty 1.5 exEquityInstrument]
```

The issuer then lifecycles each event individually, to generate two alternative lifecycling effects:

```
  -- Lifecycle dividend option
  (_, [effectCashCid]) <- submit issuer do
    exerciseCmd distributionRuleCid Lifecycle.Evolve with
      observableCids = []
      eventCid = distributionEventCashCid
      instrument = cumEquityInstrument

  (_, [effectStockCid]) <- submit issuer do
    exerciseCmd distributionRuleCid Lifecycle.Evolve with
      observableCids = []
      eventCid = distributionEventStockCid
      instrument = cumEquityInstrument
```

The investor can then claim one or the other:

```
  -- The investor chooses the stock dividend
  result <- submitMulti [investor] [publicParty] do
    exerciseCmd claimRuleCid Claim.ClaimEffect with
      claimer = investor
      holdingCids = [investorEquityCid]
      effectCid = effectStockCid
      batchId = Id "DividendOptionSettlement"
```

When this is settled, the investor's holding is consumed, which prevents the investor from receiving more than one of the dividend options.

1.19.8.6 Rights Issue

In order to raise money, a company may decide to issue new shares and give current shareholders the right (but not the obligation) to purchase those additional shares at a discounted price. This can be modeled using two components:

An option instrument, which describes the economic terms of the rights a shareholder receives. For example, this could be a European option with a strike price below the current spot price, and a maturity three weeks in the future. The page on the *Option Instrument package* describes how to create a physically settled European option.
The `DeclareDistribution` choice to distribute the above option instrument in the correct proportion (e.g. 3 option contracts for each 10 shares held). This can be done in the same way as the Bonus Issue example described earlier, just change the `perUnitDistribution` line to distribute the option instrument you created above.

When current shareholders receive the option instrument they can typically choose between:

1. Exercising the option. This corresponds to a Rights Subscription (described in more detail in the next section below).
2. Choosing not to exercise the option. The option will expire worthless.
3. Selling the option. This is not always possible, it depends on the terms of the rights issue. *Getting Started: Settlement* describes how this could be done.

1.19.8.7 Rights Subscription

Investors that decide to purchase those additional shares (subscribe to the stock issuance) can elect to exercise their right (a call option), either in parts or in whole. Sometimes, it is also possible to to apply for excess subscription. For example, an investor would like to subscribe for 150 shares but has regular rights for only 100 shares. In that case, the investor would:

Exercise the call option in whole to subscribe for the guaranteed part (100 shares).
Write a put option for the excess part (50 shares). The issuer could then exercise this in part or in whole.

More details on the Rights Issue and Subscription process are described in Instrument/Equity/Test/RightsIssue.daml , in particular how to define and process an *Election*.

1.19.8.8 Stock split

A stock split is when a company increases its number of shares. For example, a 2-for-1 stock split means that a shareholder will have two shares after the split for every share held before the split. This is modeled using the `DeclareStockSplit` choice, which has an `adjustmentFactor` argument.

The `DeclareStockSplit` choice creates a *Replacement Event*, which allows you to replace units of an instrument with another instrument (or a basket of other instruments). Consequently, this interface can also be used for other types of corporate actions (for example, see the *merger* scenario below).

The workflow for a stock split is quite similar to that of a dividend above. We start by defining the instrument before and after the lifecycle event:

```
  preEquityInstrument <- originateEquity issuer issuer "INST-1" "0" "AAPL" pp now
  postEquityInstrument <- originateEquity issuer issuer "INST-1" "1" "AAPL" [] now
```

We create a replacement rule for the stock split:

```
  -- Create lifecycle rule
  replacementRuleCid <- toInterfaceContractId @Lifecycle.I <$> submit issuer do
    createCmd Replacement.Rule with
      providers = S.singleton issuer
      lifecycler = issuer
      observers = S.singleton publicParty
      id = Id "LifecycleRule"
      description = "Rule to lifecycle an instrument following a replacement event
↪"
```

We also need a replacement event. For a 2-for-1 stock split, the `adjustmentFactor` is 1/2 = 0.5:

```
  -- Create stock split event
  replacementEventCid <-
    Instrument.submitExerciseInterfaceByKeyCmd @Equity.I [issuer] []
↪preEquityInstrument
      Equity.DeclareStockSplit with
        id = Id $ "APPL - " <> show now
        description = "Stocksplit"
        effectiveTime = now
        newInstrument = postEquityInstrument
        adjustmentFactor = 0.5
```

This allows the issuer to lifecycle the instrument:

```
  -- Lifecycle stock split
  (_, [effectCid]) <- submit issuer do
    exerciseCmd replacementRuleCid Lifecycle.Evolve with
      observableCids = []
      eventCid = replacementEventCid
      instrument = preEquityInstrument
```

This results in a lifecycle effect, which can be settled (similar to the *dividend* scenario above).

Reverse Stock Split

The stock split described above increases the number of shares available. Alternatively, a company may also decide to *decrease* the number of shares. This is referred to as *reverse stock split* or *stock consolidation*.

The `DeclareStockSplit` choice supports this as well. For example, for a 1-for-10 reverse split, modify the `adjustmentFactor` to 10/1 = 10.0 in the example above.

1.19.8.9 Merger

The merger scenario models the case when one company acquires another company and pays for it using its own shares. This is modeled using the `DeclareReplacement` choice, which also uses the *Replacement Event* (like the *stock split* scenario above). This is a mandatory exchange offer: no election is required (or possible) by the shareholder.

We start by defining the instrument before and after the merger. Shares of company ABC are being replaced by shares of company XYZ:

```
  mergingInstrument <- originateEquity merging merging "INST-1" "0" "ABC" pp now
  mergedInstrument <- originateEquity merged merged "INST-2" "0" "XYZ" pp now
```

We create a replacement rule for the merger:

```
  -- Create lifecycle rules
  replacementRuleCid <- toInterfaceContractId @Lifecycle.I <$> submit merging do
    createCmd Replacement.Rule with
      providers = S.singleton merging
      lifecycler = merging
      observers = S.singleton publicParty
      id = Id "LifecycleRule"
      description = "Rule to lifecycle an instrument following a replacement event
↪"
```

We also need a replacement event. Two shares of ABC are replaced by one share of XYZ, so the factor used in `perUnitReplacement` is 0.5:

```
  -- Create replacement event
  -- perUnitReplacement is an arbitrary list of instruments, so the investor can
↪also receive a
  -- combination of shares and cash.
  replacementEventCid <-
    Instrument.submitExerciseInterfaceByKeyCmd @Equity.I [merging] []
↪mergingInstrument
      Equity.DeclareReplacement with
        id = Id $ "ABC merge - " <> show now
        description = "Merge"
        effectiveTime = now
        perUnitReplacement = [qty 0.5 mergedInstrument]
```

This allows the issuer to lifecycle the instrument:

```
  -- Lifecycle replacement event
  (_, [effectCid]) <- submit merging do
    exerciseCmd replacementRuleCid Lifecycle.Evolve with
      eventCid = replacementEventCid
      observableCids = []
      instrument = mergingInstrument
```

This results in a lifecycle effect, which can be settled as usual.

1.19.8.10 Frequently Asked Questions

How do I transfer or trade an Equity?

When you have created a holding on an Equity instrument this can be transferred to another party. This is described in the *Getting Started: Transfer* tutorial.

In order to trade an Equity (transfer it in exchange for cash) you can also initiate a delivery versus payment with atomic settlement. This is described in the *Getting Started: Settlement* tutorial.

How do I process dividend payments for an Equity?

On the dividend payment date, the issuer will need to lifecycle the Equity. This will result in a lifecycle effect for the dividend, which can be cash settled. This is described in detail in the *Lifecycling* and the *Intermediated Lifecycling* tutorials (depending on what kind of settlement you need).

1.19.9 How To Use the Option Extension Package

To follow the script used in this tutorial, you can clone the Daml Finance repository. In particular, the Option test folder Instrument/Option/Test/ is the starting point of this tutorial.

1.19.9.1 How To Create an Option Instrument

In order to create an option instrument, you first have to decide what type of option you need. The *option extension package* currently supports the following types of options:

European Options

European options give the holder the right, but not the obligation, to buy (in case of a *call*) or to sell (in case of a *put*) the underlying asset at predetermined *strike* price on a specific *expiry* date in the future.

Daml Finance supports two types of European Options:

Physically settled European Option

The *EuropeanPhysical* instrument models physically settled call or put options.

There are two important characteristics of this instrument:

1. *physical settlement*: option holders that choose to exercise will buy (in case of a *call*) or sell (in case of a *put*) the underlying asset at the predetermined *strike* price. Since this option instrument is physically settled, it means that the underlying asset will change hands.
2. *manual exercise*: This option instrument is not automatically exercised. Instead, the option holder must manually decide whether or not to exercise. This is done by making an *Election*.

As an example, consider an option instrument that gives the holder the right to buy AAPL stock at a given price. This example is taken from Instrument/Option/Test/EuropeanPhysical.daml , where all

the details are available. Also, Check out the *Election based lifecycling tutorial* for more details on how how to define and process an *Election* in practice.

You start by defining the terms:

```
    strike = 50.0
    expiryDate = date 2019 May 15
```

Now that the terms have been defined, you can create the option instrument:

```
    let
      instrument = InstrumentKey with
        issuer
        depository
        id = Id label
        version = "0"

    cid <- submitMulti [issuer] [publicParty] do
      exerciseCmd europeanPhysicalOptionFactoryCid EuropeanPhysical.Create with
        european = EuropeanPhysical.European with
          instrument
          description
          expiryDate
          optionType
          strike
          referenceAsset
          ownerReceives
          currency
          lastEventTimestamp
          prevElections = []
        observers = M.fromList observers
```

Once this is done, you can create a holding on it using *Account.Credit*.

Cash-settled European Option

The *EuropeanCash* instrument models cash-settled, auto-exercising call or put options. They are similar to the *EuropeanPhysical* instrument described above, but there are two important differences:

1. *cash settlement*: This means that the underlying asset will not change hands. Instead, a cash settlemement will take place (if the option is exercised).
2. *automatic exercise*: This option instrument is automatically exercised. No manual *Election* is required by the holder.

As an example, consider an option instrument that gives the holder the right to buy AAPL stock at a given price. This example is taken from Instrument/Option/Test/EuropeanCash.daml , where all the details are available.

You start by defining the terms:

```
    strike = 50.0
    expiryDate = date 2019 May 15
    referenceAssetId = "AAPL-CLOSE"
```

Now that the terms have been defined, you can create the option instrument:

```
    let
      instrument = InstrumentKey with
        issuer
        depository
        id = Id label
        version = "0"

    cid <- submitMulti [issuer] [publicParty] do
      exerciseCmd europeanCashOptionFactoryCid EuropeanCash.Create with
        european = EuropeanCash.European with
          instrument
          description
          expiryDate
          optionType
          strike
          referenceAssetId
          ownerReceives
          currency
          lastEventTimestamp
        observers = M.fromList observers
```

Once this is done, you can create a holding on it using *Account.Credit*.

If the close price of AAPL on the expiry date is above the *strike* price, the option holder would profit from exercising the option and buying the stock at the strike price. The value of the option would be *spot - strike*. Since this option type is cash-settled, this amount would be paid in the option currency after lifecycling.

On the other hand, if the close price of AAPL is below the *strike* price, the option would expire worthless.

This option instrument is automatically exercised. This means that the decision whether or not to exercise is done automatically by comparing the *strike* price to an observation of the close price. For this to work, you need to define an *Observation* as well:

```
  let observations = M.fromList [(dateToDateClockTime $ date 2019 May 15, 48.78)]
  observableCid <- toInterfaceContractId <$> submit issuer do
    createCmd Observation with
      provider = issuer; id = Id referenceAssetId; observations; observers = M.
↪empty
```

Barrier Option

The *BarrierEuropeanCash* instrument models barrier options. They are similar to the *EuropeanCash* instrument described above, but also contain a barrier that is used to activate (or, alternatively, knock out) the option. The *BarrierTypeEnum* describes which barrier types are supported.

As an example, consider an option instrument that gives the holder the right to buy AAPL stock at a given price. However, if AAPL ever trades at or below a given barrier level, the option is knocked out (which means that it expires worthless). In other words, this describes a *DownAndOut* option. This example is taken from Instrument/Option/Test/BarrierEuropeanCash.daml , where all the details are available.

You start by defining the terms:

```
    barrier = 30.0
    barrierType = DownAndOut
    barrierStartDate = date 2019 Jan 20
    strike = 40.0
    expiryDate = date 2019 May 15
    referenceAssetId = "AAPL-CLOSE"
```

Now that the terms have been defined, you can create the option instrument:

```
    let
      instrument = InstrumentKey with
        issuer
        depository
        id = Id label
        version = "0"

    cid <- submitMulti [issuer] [publicParty] do
      exerciseCmd barrierOptionFactoryCid BarrierEuropeanCash.Create with
        barrierEuropean = BarrierEuropean with
          instrument
          description
          expiryDate
          optionType
          strike
          barrier
          barrierType
          barrierStartDate
          referenceAssetId
          ownerReceives
          currency
          lastEventTimestamp
        observers = M.fromList observers
```

Once this is done, you can create a holding on it using *Account.Credit*.

Compared to the *EuropeanCash option* this instrument needs to be lifecycled not only at expiry but also during its lifetime in case of a barrier hit. This is done in the same way as lifecycling at maturity, i.e. an *Observation* is provided for the reference asset identifier, containing the date and the underlying price.

Dividend Option

The *Dividend* instrument models physically settled, manually exercised dividend options. For reference, a dividend option gives the holder the right to choose one out of several dividend payouts, on a specific *expiry* date in the future. The following payout types are supported:

1. *Cash*: The dividend is paid in cash. This a mandatory option. In addition, the issue can offer:
2. *Shares*: The dividend is paid in shares. To the investor this is similar to a *Bonus Issue*.
3. *CashFx*: The dividend is paid in cash in a foreign currency.

As an example, consider an option instrument that gives the holder the right to choose to receive AAPL dividends either as cash, shares or cash in a foreign currency (EUR). This example is taken from Instrument/Option/Test/Dividend.daml , where all the details are available.

You start by defining the terms:

```
    expiryDate = date 2019 May 15
    cashQuantity = qty 0.19 cashInstrument
    sharesQuantity = Some $ qty 0.0041 aaplInstrument
    fxQuantity = Some $ qty 0.17 eurInstrument
```

Now that the terms have been defined, you can create the option instrument:

```
    let
      instrument = InstrumentKey with
        issuer
        depository
        id = Id label
        version = "0"

    cid <- submitMulti [issuer] [publicParty] do
      exerciseCmd dividendOptionFactoryCid Dividend.Create with
        dividend = Dividend with
          instrument
          description
          expiryDate
          cashQuantity
          sharesQuantity
          fxQuantity
          lastEventTimestamp
          prevElections = []
        observers = M.fromList observers
```

Once this is done, you can create a holding on it using *Account.Credit*.

On the expiry date, the option holder will make an *Election* out of the available choices. The lifecycling of this option works in the same way as for *physically settled European options*.

1.19.9.2 Frequently Asked Questions

How do I transfer or trade an option?

When you have created a holding on an option instrument this can be transferred to another party. This is described in the *Getting Started: Transfer* tutorial.

In order to trade an option (transfer it in exchange for cash) you can also initiate a delivery versus payment with atomic settlement. This is described in the *Getting Started: Settlement* tutorial.

How do I calculate settlement payments for an option?

On the expiry date, the issuer will need to lifecycle the European option. This will result in a lifecycle effect for the payoff, which can be cash settled. This is described in detail in the *Lifecycling* and the *Intermediated Lifecycling* tutorials.

1.19.10 How To Use the Swap Instrument Packages

To follow the code snippets used in this page in Daml Studio, you can clone the Daml Finance repository and run the scripts included in the Instrument/Swap/Test/ folder.

1.19.10.1 Prerequisites

Swap instruments share many similarities with Bond instruments. This page builds on the page for the *Bond Instruments*. Please, check it out before reading the Swap specifics below.

1.19.10.2 How To Create a Swap Instrument

There are different types of swaps, which differ both in the way regular payments are defined and whether notional is exchanged. In order to create a swap instrument, you first have to decide what type of swap you need. The *swap instrument package* currently supports the following types of swaps:

Interest Rate

Interest rate swap is the type of swap that shares most similarities with a bond. It has two legs: one which pays a fix rate and another one which pays a floating rate. These rates are paid at the end of every payment period.

As an example, we will create a swap instrument paying Libor 3M vs a 2.01% p.a. with a 3M payment period. This example is taken from Instrument/Swap/Test/InterestRate.daml , where all the details are available.

We start by defining the terms:

```
let
  issueDate = date 2019 Jan 16
  firstPaymentDate = date 2019 Feb 15
  maturityDate = date 2019 May 15
  referenceRateId = "USD/LIBOR/3M"
  ownerReceivesFix = False
  fixRate = 0.0201
  paymentPeriod = M
  paymentPeriodMultiplier = 3
  dayCountConvention = Act360
  businessDayConvention = ModifiedFollowing
```

The floating leg depends on a reference rate, which is defined by the *referenceRateId* variable. The value of the reference rate is observed at the beginning of each payment period.

The *ownerReceivesFix* variable is used to specify whether a holding owner of this instrument receives the fix or the floating leg. This is not needed for bonds, because the regular payments are always in one direction (from the issuer to the holder). However, in the case of a swap with two counterparties A and B, we need the *ownerReceivesFix* variable to specify who receives fix and who receives floating. In this example, the holding owner receives the floating leg.

Just as for bonds, we can use these variables to create a *PeriodicSchedule*:

```
    let
      (y, m, d) = toGregorian firstCouponDate
      periodicSchedule = PeriodicSchedule with
        businessDayAdjustment =
          BusinessDayAdjustment with
            calendarIds = holidayCalendarIds
            convention = businessDayConvention
        effectiveDateBusinessDayAdjustment = None
        terminationDateBusinessDayAdjustment = None
        frequency =
          Periodic Frequency with
            rollConvention = DOM d
            period = Period with
              period = couponPeriod
              periodMultiplier = couponPeriodMultiplier
        effectiveDate = issueDate
        firstRegularPeriodStartDate = Some firstCouponDate
        lastRegularPeriodEndDate = Some maturityDate
        stubPeriodType = None
        terminationDate = maturityDate
```

Note that this instrument only has one periodic schedule, which is used for both the fixed and the floating leg. It is also used for both the calculation period (to determine which floating rate to be used) and the payment period (to determine when payments are done). The *FpML swap template* below offers more flexibility here. It has individual schedules, both for the fixed/floating leg and for the calculation/payment periods. That would allow you to specify whether payments should be made e.g. after each calculation period or only after every second calculation period.

Now that we have defined the terms we can create the swap instrument:

```
    let
      instrument = InstrumentKey with
        issuer
        depository
        id = Id label
        version = "0"

    cid <- submitMulti [issuer] [publicParty] do
      exerciseCmd interestRateSwapFactoryCid InterestRate.Create with
        interestRate = InterestRate with
          instrument
          description
          periodicSchedule
          holidayCalendarIds
          calendarDataProvider
          dayCountConvention
          referenceRateId
          fixRate
          ownerReceivesFix
          currency
          lastEventTimestamp
        observers = M.fromList observers
```

Once this is done, you can create a holding on it using *Account.Credit*. The owner of the holding receives the floating leg (and pays the fix leg).

Currency

Currency swaps are quite similar to interest rate swaps, except that the two legs are in different currencies. Consequently, we need to create two cash instruments:

```
cashInstrument <- originate custodian issuer "USD" "US Dollars" observers now
foreignCashInstrument <- originate custodian issuer "EUR" "Euro" observers now
```

In the swap template they are referred to as *base currency* and *foreign currency*.

Here is an example of a fix vs fix currency swap: 3% p.a. in USD vs 2% p.a. in EUR with payments every 3M:

```
let
  issueDate = date 2019 Jan 16
  firstPaymentDate = date 2019 Feb 15
  maturityDate = date 2019 May 15
  ownerReceivesBase = False
  baseRate = 0.03
  foreignRate = 0.02
  fxRate = 1.1
  paymentPeriod = M
  paymentPeriodMultiplier = 3
  dayCountConvention = Act360
  businessDayConvention = ModifiedFollowing
```

In this example, the holding owner receives the foreign currency leg.

In order to calculate the interest rate payments, a notional is required in each currency. The quantity of the holding refers to the notional of the base currency. The notional of the foreign currency is defined as the quantity of the holding multiplied by the specified *fxRate*.

Note that this template is limited to fixed rates. It also does not support exchange of notionals. If you need floating rates or exchange of notionals, please use the *FpML swap template* below. It supports both of those features.

Here is how we create the currency swap instrument, using the two currencies defined above:

```
  let
    instrument = InstrumentKey with
      issuer
      depository
      id = Id label
      version = "0"

  cid <- submitMulti [issuer] [publicParty] do
    exerciseCmd currencySwapFactoryCid Currency.Create with
      currencySwap = CurrencySwap with
        instrument
        description
        periodicSchedule
        holidayCalendarIds
        calendarDataProvider
        dayCountConvention
        ownerReceivesBase
        baseRate
        foreignRate
```

(continues on next page)

(continued from previous page)

```
          baseCurrency
          foreignCurrency
          fxRate
          lastEventTimestamp
        observers = M.fromList observers
```

Once the instrument is created, you can create a holding on it. In our example, it the owner of the holding receives the foreign currency leg (and pays the base currency leg).

Foreign Exchange

Despite the similarities in name, *foreign exchange swaps* (or FX swaps) are quite different from currency swaps. An FX swap does not pay or receive interest. Instead, the two legs define an initial FX transaction and a final FX transaction. Each transaction requires an FX rate and a transaction date, which are predetermined between the counterparties.

The FX transactions involve two currencies. In the swap template these are referred to as *base currency* and *foreign currency*. The convention is that the holding owner receives the foreign currency in the initial transaction (and pays it in the final transaction).

Here is an example of an USD vs EUR FX swap. First, we define the two cash instruments:

```
cashInstrument <- originate custodian issuer "USD" "US Dollars" observers now
foreignCashInstrument <- originate custodian issuer "EUR" "Euro" observers now
```

Then, we define the transaction dates and FX rates:

```
let
  issueDate = date 2019 Jan 16
  firstPaymentDate = date 2019 Feb 15
  maturityDate = date 2019 May 15
  firstFxRate = 1.1
  finalFxRate = 1.2
```

The *firstPaymentDate* variable defines the date of the initial FX transaction. Generally, this is on the issue date or shortly afterwards.

Finally, we create the FX swap instrument:

```
  let
    instrument = InstrumentKey with
      issuer
      depository
      id = Id label
      version = "0"

  cid <- submitMulti [issuer] [publicParty] do
    exerciseCmd fxSwapFactoryCid ForeignExchange.Create with
      foreignExchange = ForeignExchange with
        instrument
        description
        baseCurrency
        foreignCurrency
        firstFxRate
```

(continues on next page)

(continued from previous page)

```
          finalFxRate
          issueDate
          firstPaymentDate
          maturityDate
          lastEventTimestamp
        observers = M.fromList observers
```

Once the instrument is created, you can create a holding on it. The owner of the holding receives the foreign currency in the initial transaction. In the final transaction the sides are reversed.

Credit Default

A *credit default swap* (CDS) pays a protection amount in case of a credit default event, in exchange for a fix rate at the end of every payment period. The protection amount is defined as *1-recoveryRate*. The *recoveryRate* is defined as the amount recovered when a borrower defaults, expressed as a percentage of notional.

If a credit event occurs, the swap expires after the protection amount has been paid, i.e., no more rate payments are required afterwards.

Here is an example of a CDS that pays *1-recoveryRate* in the case of a default on TSLA bonds:

```
issueDate = date 2019 Jan 16
firstPaymentDate = date 2019 Feb 15
maturityDate = date 2019 May 15
defaultProbabilityReferenceId = "TSLA-DEFAULT-PROB"
recoveryRateReferenceId = "TSLA-RECOVERY-RATE"
ownerReceivesFix = False
fixRate = 0.0201
paymentPeriod = M
paymentPeriodMultiplier = 3
dayCountConvention = Act360
businessDayConvention = ModifiedFollowing
```

In our example, the issuer pays the protection leg of the swap.

As you can see in this example, two observables are required for a CDS:

1. *defaultProbabilityReferenceId*: The reference ID of the default probability observable. For example, in case of protection against a TSLA bond payment default this should be a valid reference to the TSLA default probability .
2. *recoveryRateReferenceId*: The reference ID of the recovery rate observable. For example, in case of a TSLA bond payment default with a 60% recovery rate this should be a valid reference to the TSLA bond recovery rate .

Finally, we create the CDS instrument:

```
    let
      instrument = InstrumentKey with
        issuer
        depository
        id = Id label
        version = "0"
```

(continues on next page)

(continued from previous page)

```
    cid <- submitMulti [issuer] [publicParty] do
      exerciseCmd cdsFactoryCid CreditDefault.Create with
        creditDefault = CreditDefault with
          instrument
          description
          periodicSchedule
          holidayCalendarIds
          calendarDataProvider
          dayCountConvention
          fixRate
          ownerReceivesFix
          defaultProbabilityReferenceId
          recoveryRateReferenceId
          currency
          lastEventTimestamp
        observers = M.fromList observers
```

Once the instrument is created, you can create a holding on it. In our example, the owner of the holding receives the protection leg (and pays the fix leg).

Asset

An *asset swap* is a general type of swap with two legs: one which pays a fix rate and another one which pays the performance of an asset. It can be used to model:

- equity swaps
- some types of commodity swaps (of the form *performance vs rate*)
- other swaps with the same payoff on other asset types.

Here is an example of an asset swap that pays AAPL total return vs 2.01% fix p.a., payment every 3M:

```
  let
    issueDate = date 2019 Jan 16
    firstPaymentDate = date 2019 Feb 15
    maturityDate = date 2019 May 15
    referenceAssetId = "AAPL-CLOSE-ADJ"
    ownerReceivesFix = False
    fixRate = 0.0201
    paymentPeriod = M
    paymentPeriodMultiplier = 3
    dayCountConvention = Act360
    businessDayConvention = ModifiedFollowing
```

In our example, the issuer pays the asset leg of the swap.

One observable is required: *referenceAssetId*. The template calculates the performance for each payment period using this observable. Performance is calculated from the start date to the end date of each payment period. The reference asset Observable needs to contain the appropriate type of fixings:

- *unadjusted* fixings in case of a *price return* asset swap
- *adjusted* fixings in case of a *total return* asset swap

Finally, we create the asset swap instrument:

```
    let
      instrument = InstrumentKey with
        issuer
        depository
        id = Id label
        version = "0"

    cid <- submitMulti [issuer] [publicParty] do
      exerciseCmd assetSwapFactoryCid Asset.Create with
        asset = Asset with
          instrument
          description
          periodicSchedule
          holidayCalendarIds
          calendarDataProvider
          dayCountConvention
          fixRate
          ownerReceivesFix
          referenceAssetId
          currency
          lastEventTimestamp
        observers = M.fromList observers
```

Once this is done, you can create a holding on it. The owner of the holding receives the asset leg (and pays the fix leg).

FpML

Unlike the other swap types above, the *FpML swap* template is not a new type of payoff. Instead, it allows you to input other types of swaps using the FpML schema. Currently, interest rate swaps and currency swaps are supported. The template can quite easily be extended to FX swaps.

Specifically, it allows you to specify one swapStream object for each leg of the swap.

We start by defining the general terms:

```
  let
    issueDate = date 2022 Sep 14
    firstRegularPeriodDate = date 2022 Sep 20
    lastRegularPeriodDate = date 2023 Jun 20
    maturityDate = date 2023 Sep 14
    firstRegularPeriodDateFixLeg = date 2022 Sep 20
    lastRegularPeriodDateFixLeg = firstRegularPeriodDateFixLeg
    referenceRateId = "USD/LIBOR/3M"
    referenceRateOneMonthId = "USD/LIBOR/1M"
    fixRate = 0.02
    paymentPeriod = Regular M
    paymentPeriodMultiplier = 3
    dayCountConvention = Act360
    businessDayConvention = ModifiedFollowing
    issuerPartyRef = "Counterparty"
    clientPartyRef = "ExecutingParty"
```

The *issuerPartyRef* and the *clientPartyRef* variables are used to specify who pays each leg (see *payerPartyReference* below).

The fixed leg of the swap can now be defined using Daml data types that correspond to the swapStream schema:

```
    swapStreamFixedLeg = SwapStream with
      payerPartyReference = clientPartyRef
      receiverPartyReference = issuerPartyRef
      calculationPeriodDates = CalculationPeriodDates with
        id = "fixedLegCalcPeriodDates"
        effectiveDate = AdjustableDate with
          unadjustedDate = issueDate
          dateAdjustments = BusinessDayAdjustments with
            businessDayConvention = NoAdjustment
            businessCenters = []
        terminationDate = AdjustableDate with
          unadjustedDate = maturityDate
          dateAdjustments = BusinessDayAdjustments with
            businessDayConvention = ModifiedFollowing
            businessCenters = holidayCalendarIds
        calculationPeriodDatesAdjustments = CalculationPeriodDatesAdjustments with
          businessDayConvention = ModifiedFollowing
          businessCenters = holidayCalendarIds
        firstPeriodStartDate = None
        firstRegularPeriodStartDate = Some firstRegularPeriodDateFixLeg
        lastRegularPeriodEndDate = Some lastRegularPeriodDateFixLeg
        calculationPeriodFrequency = CalculationPeriodFrequency with
          periodMultiplier = 1
          period = Regular Y
          rollConvention = DOM 20
      paymentDates = PaymentDates with
        calculationPeriodDatesReference = "fixedLegCalcPeriodDates"
        paymentFrequency = PaymentFrequency with
          periodMultiplier = 1
          period = Regular Y
        firstPaymentDate = Some firstRegularPeriodDateFixLeg
        lastRegularPaymentDate = Some lastRegularPeriodDateFixLeg
        payRelativeTo = CalculationPeriodEndDate
        paymentDatesAdjustments = BusinessDayAdjustments with
          businessDayConvention = ModifiedFollowing
          businessCenters = holidayCalendarIds
        paymentDaysOffset = None
      resetDates = None
      calculationPeriodAmount = CalculationPeriodAmount with
        calculation = Calculation with
          notionalScheduleValue = NotionalSchedule_Regular NotionalSchedule with
            id = "fixedLegNotionalSchedule"
            notionalStepSchedule = NotionalStepSchedule with
              initialValue = 1000000.0
              step = []
              currency = "USD"
          rateTypeValue = RateType_Fixed FixedRateSchedule with
            initialValue = fixRate
            step = []
          dayCountFraction = dayCountConvention
          compoundingMethodEnum = None
      stubCalculationPeriodAmount = None
      principalExchanges = None
```

As you can see, the *Daml SwapStream data type* matches the swapStream FpML schema. Please note

that the actual parsing from FpML to Daml is not done by this template. It has to be implemented on the client side.

Similarly, the floating leg of the swap is defined like this:

```
    swapStreamFloatingLeg = SwapStream with
      payerPartyReference = issuerPartyRef
      receiverPartyReference = clientPartyRef
      calculationPeriodDates = CalculationPeriodDates with
        id = "floatingLegCalcPeriodDates"
        effectiveDate = AdjustableDate with
          unadjustedDate = issueDate
          dateAdjustments = BusinessDayAdjustments with
            businessDayConvention = NoAdjustment
            businessCenters = []
        terminationDate = AdjustableDate with
          unadjustedDate = maturityDate
          dateAdjustments = BusinessDayAdjustments with
            businessDayConvention = ModifiedFollowing
            businessCenters = holidayCalendarIds
        calculationPeriodDatesAdjustments = CalculationPeriodDatesAdjustments with
          businessDayConvention = ModifiedFollowing
          businessCenters = holidayCalendarIds
        firstPeriodStartDate = None
        firstRegularPeriodStartDate = Some firstRegularPeriodDate
        lastRegularPeriodEndDate = Some lastRegularPeriodDate
        calculationPeriodFrequency = CalculationPeriodFrequency with
          periodMultiplier = paymentPeriodMultiplier
          period = paymentPeriod
          rollConvention = DOM 20
      paymentDates = PaymentDates with
        calculationPeriodDatesReference = "floatingLegCalcPeriodDates"
        paymentFrequency = PaymentFrequency with
          periodMultiplier = paymentPeriodMultiplier
          period = paymentPeriod
        firstPaymentDate = Some firstRegularPeriodDate
        lastRegularPaymentDate = Some lastRegularPeriodDate
        payRelativeTo = CalculationPeriodEndDate
        paymentDatesAdjustments = BusinessDayAdjustments with
          businessDayConvention = ModifiedFollowing
          businessCenters = holidayCalendarIds
        paymentDaysOffset = None
      resetDates = Some ResetDates with
        calculationPeriodDatesReference = "floatingLegCalcPeriodDates"
        resetRelativeTo = CalculationPeriodStartDate
        fixingDates = FixingDates with
          periodMultiplier = -2
          period = D
          dayType = Some Business
          businessDayConvention = NoAdjustment
          businessCenters = fixingHolidayCalendarId
        resetFrequency = ResetFrequency with
          periodMultiplier = paymentPeriodMultiplier
          period = paymentPeriod
        resetDatesAdjustments = ResetDatesAdjustments with
          businessDayConvention = ModifiedFollowing
          businessCenters = holidayCalendarIds
```

(continues on next page)

(continued from previous page)

```
    calculationPeriodAmount = CalculationPeriodAmount with
      calculation = Calculation with
        notionalScheduleValue = NotionalSchedule_Regular NotionalSchedule with
          id = "floatingLegNotionalSchedule"
          notionalStepSchedule = NotionalStepSchedule with
            initialValue = 1000000.0
            step = []
            currency = "USD"
        rateTypeValue = RateType_Floating FloatingRateCalculation with
          floatingRateIndex = referenceRateId
          indexTenor = Some Period with
            periodMultiplier = paymentPeriodMultiplier
            period = M
          spreadSchedule = [SpreadSchedule with initialValue = 0.005]
          finalRateRounding = None
        dayCountFraction = dayCountConvention
        compoundingMethodEnum = Some Straight
    stubCalculationPeriodAmount = Some StubCalculationPeriodAmount with
      calculationPeriodDatesReference = "floatingLegCalcPeriodDates"
      initialStub = Some $ StubValue_StubRate 0.015
      finalStub = Some $ StubValue_FloatingRate
        [ StubFloatingRate with
            floatingRateIndex=referenceRateOneMonthId
            indexTenor = Some (Period with period = M; periodMultiplier = 1)
        , StubFloatingRate with
            floatingRateIndex = referenceRateId
            indexTenor = Some (Period with period = M; periodMultiplier = 3)
        ]
    principalExchanges = None
```

There are three main ways to define which interest rate should be used for a stub period. They are all included in the fix or floating leg above, either in the inital or in the final stub period. In short, it depends on the content of *StubCalculationPeriodAmount*:

1. *None*: No special stub rate is provided. Instead, use the same rate as was specified in the corresponding *Calculation*.
2. Specific *stubRate*: Use this specific fix rate.
3. Specific *floatingRate*: Use this specific floating rate (if one rate is provided). If two rates are provided: use linear interpolation between the two rates.

Finally, we create the FpML swap instrument:

```
  let
    instrument = InstrumentKey with
      issuer
      depository
      id = Id label
      version = "0"

  cid <- submitMulti [issuer] [publicParty] do
    exerciseCmd fpmlSwapFactoryCid Fpml.Create with
      fpml = Fpml with
        instrument
        description
        swapStreams
```

(continues on next page)

(continued from previous page)

```
          issuerPartyRef
          currencies
          calendarDataProvider
          lastEventTimestamp
        observers = M.fromList observers
```

Once this is done, you can create a holding on it. In this particular example trade, the notional is specified in the FpML instrument. This means that you would only book a unit holding (quantity=1.0) on the instrument.

1.19.10.3 Frequently Asked Questions

Why do the swaps have an issuer?

In the case of bonds, the instrument has a well-defined issuer. This is not necessarily the case for swaps, where two counterparties A and B swap the payments associated with each leg. However, in practice one of the counterparties is often a swap dealer, who shares some of the characteristics of a bond issuer. For the purpose of lifecycling in Daml Finance, we require one of the counterparties to take the role as issuer. This counterparty will serve as calculation agent and provide the observables required to calculate the swap payments.

The documentation of the Daml Finance asset model contains an *OTC swap example*.

1.19.11 How to use the Generic Instrument packages

The *Generic Instrument* provides a flexible framework to model and lifecycle custom payoffs in Daml Finance. It encapsulates the *Contingent Claims* library, which gives us the tools to model the economic terms of the payoff.

To follow the code snippets used in this page in Daml Studio, you can clone the Daml Finance repository and run the scripts in the Instrument/Generic/Test/Intermediated/BondCoupon.daml and Instrument/Generic/Test/EuropeanOption.daml files.

The Generic Instrument and the Contingent Claims library are introduced in the *Payoff Modeling tutorial*, which we encourage you to check out.

1.19.11.1 How to create a Generic Instrument

Define the Claim of a Bond

Consider a fixed rate bond which pays a 4% coupon *per annum* with a coupon period of 6 months. Assume that there are two coupons remaining until maturity: one to be paid today and one to be paid in 180 days. This can be modeled in the following way:

```
    let
      today = toDateUTC now
      expiry = addDays today 180
      bondLabel = "ABC.DE 4% p.a. " <> show expiry <> " Corp"
      claims = mapClaimToUTCTime $ andList
```

(continues on next page)

(continued from previous page)

```
      [ when (at today) $ scale (Const 0.02) $ one cashInstrument
      , when (at expiry) $ scale (Const 0.02) $ one cashInstrument
      , when (at expiry) $ scale (Const 1.0) $ one cashInstrument
      ]
```

Now that we have specified the economic terms of the payoff we can create a generic instrument:

```
  instrument <- originateGeneric csd issuer bondLabel "Bond" now claims pp now
```

On every coupon payment date, the issuer will need to lifecycle the instrument. This will result in a lifecycle effect for the coupon, which can be then be claimed and settled. This process is described in detail in *Getting Started: Lifecycling*.

Define the Claim of a European Option

Alternatively, if you want to model a European Option instead, you can define the claim as follows

```
let
  exerciseClaim = scale (Observe spot - Const strike) $ one ccy
  option = european maturity exerciseClaim
```

This uses the *european* builder function, which is included in *Contingent Claims*.

Compared to the bond, where the passage of time results in a coupon payment being due, the option instrument requires a manual *Election*: the holder of the instrument holding needs to choose whether or not to exercise the option. How this is done is described in the next section.

1.19.11.2 How to lifecycle a Generic Instrument

Election based lifecycling of Contingent Claims based instruments

We describe how to lifecycle an option instrument (which can be *Exercised* or *Expired*), but the same concepts apply to other Election based instruments (for example, a callable bond that can be *Called* or *NotCalled*). A similar workflow is also applicable to some of the instruments available in the Bond, Swap, and Option packages.

First, an Election factory is created:

```
-- Create election offers to allow holders to create elections
electionFactoryCid <- submit broker do
  toInterfaceContractId <$> createCmd Election.Factory with
    provider = broker
    observers = M.fromList pp
```

Then, election offers are created for the different election choices that are available. Specifically, for option instruments, an election offer to *exercise* is created:

```
exerciseElectionFactoryCid <- submit broker do
  createCmd ElectionOffer with
    provider = broker
    id = Id "EXERCISE"
```

(continues on next page)

(continued from previous page)

```
    description = "OPTION-AAPL - Exercise"
    claim = "EXERCISE"
    observers = S.singleton publicParty
    instrument = genericInstrument
    factoryCid = electionFactoryCid
```

Similarly, an election offer to *expire* the option is also created:

```
expireElectionFactoryCid <- submit broker do
  createCmd ElectionOffer with
    provider = broker
    id = Id "EXPIRE"
    description = "OPTION-AAPL - Expire"
    claim = "EXPIRE"
    observers = S.singleton publicParty
    instrument = genericInstrument
    factoryCid = electionFactoryCid
```

Assuming the investor wants to exercise the option, an election candidate contract is created. In order to do this, the investor presents a holding for which an election should be made, and also specifies the amount that this election applies to. This amount cannot exceed the quantity of the holding:

```
-- One cannot exercise for more units than they own
submitMultiMustFail [investor1] [publicParty] do
  exerciseCmd exerciseElectionFactoryCid CreateElectionCandidate with
    elector = investor1
    electionTime = dateToDateClockTime maturity
    holdingCid = investor1GenericHoldingCid
    amount = 5000.0
```

Instead, the elected amount must be the same as the holding quantity, or lower:

```
-- Create election
exerciseOptionProposalCid <- submitMulti [investor1] [publicParty] do
  exerciseCmd exerciseElectionFactoryCid CreateElectionCandidate with
    elector = investor1
    electionTime = dateToDateClockTime maturity
    holdingCid = investor1GenericHoldingCid
    amount = 500.0
```

A time event is also required to indicate *when* the election is made:

```
currentTimeCid <- createDateClock (S.singleton broker) maturity S.empty
```

It is now possible to create the *Election*:

```
exerciseOptionCid <- submit broker do
  exerciseCmd exerciseOptionProposalCid ValidateElectionCandidate with
    currentTimeCid
```

Note: these templates (election offer and election candidate) are not considered a core part of the Daml Finance library. There can be different processes to create the Election, so this is rather application specific. Still, in order to showcase one way how this could be done, this workflow is included here for convenience.

The *Election* has a flag *electorIsOwner*, which indicates whether the election is on behalf of the owner of the holding. This is typically the case for options, where the option holder has the right, but not the obligation, to exercise the option. On the other hand, for callable bonds it is not the holding owner (the bond holder) who gets to decide whether the bond is redeemed early. Instead, it is the counterparty. In this case, *electorIsOwner* would be false.

A lifecycle rule is required to specify how to process the Election:

```
-- Apply election to generate new instrument version + effects
lifecycleRuleCid <- toInterfaceContractId <$> submit bank do
  createCmd Lifecycle.Rule with
    providers = S.singleton bank
    observers= M.empty
    lifecycler = broker
    id = Id "LifecycleRule"
    description = "Rule to lifecycle a generic instrument"
```

This is similar to time-based lifecycling.

Finally, it is possible to apply the Election according to the lifecycle rule provided:

```
(Some exercisedOption, [effectCid]) <- submit broker do
  exerciseCmd exerciseOptionCid Election.Apply with
    observableCids = [observableCid]
    exercisableCid = lifecycleRuleCid
```

This creates lifecycle effects, which can be claimed and settled in the usual way (as described in *Getting Started: Lifecycling*). However, the holding contract used to claim the effect must be compatible with the election that has been made: if Alice made an election and *electorIsOwner = True*, then only a holding where *owner = alice* will be accepted.

1.19.11.3 Contingent Claims

Introduction

Contingent Claims is a library for modeling financial instruments in Daml. An instrument is represented by a tree of *Claims*, which describe future cashflows between two parties as well as the conditions under which these cashflows occur.

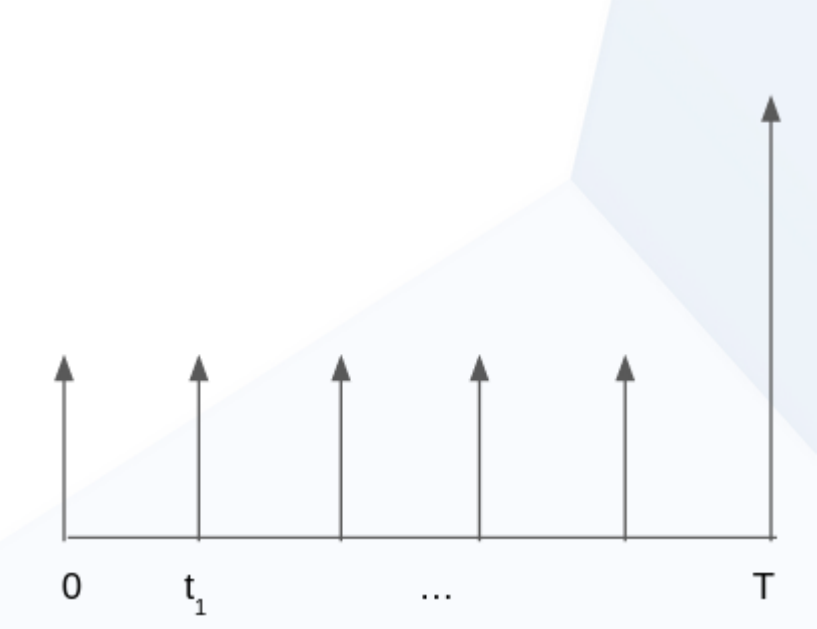
The library offers lifecycling capabilities, as well as an (experimental) valuation semantics to map a claim to a mathematical expression that can be used for no-arbitrage pricing.

Examples of how to create and lifecycle contracts using Contingent Claims can be found in the *Payoff Modeling tutorial*.

In the following we present a user guide for getting started with Contingent Claims instrument modeling. It is meant to teach the basics of the framework, but does not cover every aspect. The work is based on the papers [Cit1] and [Cit2], and we recommend that you refer to these for an in-depth understanding of how it works.

The Model

The approach taken in the papers is to model financial instruments by their cashflows. This should be familiar to anyone having taken a course in corporate finance or valuation. Let's start with an example:



The picture above represents the cashflows of a fixed-rate bond. Or alternatively, you can think of it as a mortgage, from the point of view of the bank: there are interest payments at regular intervals (the small arrows), and a single repayment of the loan at maturity (the big arrow on the right). So how do we go about modelling this?

We use the following data type, slightly simplified from *Claim*:

```
data Claim a
  = zero
  | one a
  | give (Claim a)
  | and with lhs: Claim a, rhs: Claim a
  | or with lhs: Claim a, rhs: Claim a
  | scale with k: Date -> Decimal, claim: Claim a
  | when with predicate: Date -> Bool, claim: Claim a
  | anytime with predicate : Date -> Bool, claim: Claim a
  | until with predicate : Date -> Bool, claim: Claim a
```

There are a couple of things to consider.

First note that the constructors of this data type create a tree structure. The leaf constructors are `zero` and `one a`, and the other constructors create branches (observe they call `Claim a` recursively). The constructors are just functions, and can be combined to produce complex cashflows. For example, to represent the above bond, we could write the following:

```
when (time == t_0) (scale (pure coupon) (one "USD")) `and` ...
```

Let's look at the constructors used in the above expression in more detail:

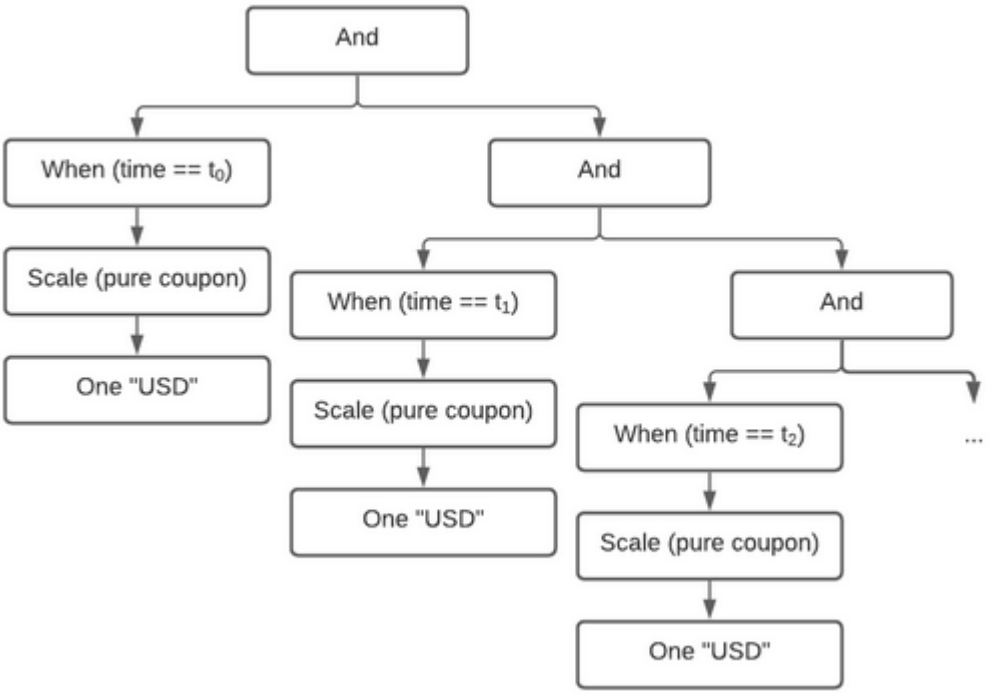
> `one "USD"` means that the acquirer of the contract receives one unit of the asset, parametrised by `a`, *immediately*. In this case we use a 3-letter ISO code to represent a currency,

but you can use your own type to represent any asset.
`scale (pure coupon)` modifies the *magnitude* of the arrow in the diagram. For example, in the diagram, the big arrow would have a distinct scale factor from the small arrows. In our example, the scale factor is constant: `pure coupon = const coupon`, however, it's possible to have a scale factor that depends on an unobserved value, such as a stock price, the weather, or any other measurable quantity.
`when (time == t_0)` tells us where along the x-axis the arrow is placed, i.e., it modifies the point in time when the claim is acquired. The convention is that this must be the first instant that the predicate (`time == t_0` in this case) is true. In our example it is a point, but again, we could have used an expression with an unknown quantity, for example `spotPrice > pure k`, and it would trigger *the first instant* that the expression becomes true.
`and` is used to chain multiple expressions together. Remember that in the `data` definition above, each constructor is a function: `and : Claim a -> Claim a -> Claim a`. We use the Daml backtick syntax to write `and` as an infix operator, for legibility.

Additionally, there are several constructors which were not used in the above example:

`zero`, used to indicate an absence of obligations. While it may not make sense to create a `zero` claim, it could, for example, result from applying a function on a tree of claims.
`give` would flip the direction of the arrows in our diagram. For example, in a swap we could use `give` to distinguish the received/paid legs.
`or` is used to give the bearer the right to choose between two different claims. This is typically used for options.
`anytime` is like `when`, except it allows the bearer to choose (vs. no choice) acquisition within a region (or timeframe), vs. a specific point in time.
`until` is used to adjust the expiration (*horizon* in [Cit1]) of a claim. Typically, it is used with `anytime` to limit aforesaid acquisition region.

The tree produced by our expression (corresponding to the cashflow figure above) looks like this:



Composition and Extensibility

Although we could model every subsequent arrow the way we did the first one, as good programmers we wish to avoid repeating ourselves. Hence, we could write functions to re-use subexpressions of the tree. But which parts should we factor out? It turns out that Finance 101 comes to the rescue again. Fixed income practitioners will typically model a fixed-rate bond as a sum of zero-coupon bonds. That's how we model them in *Claims.Util.Builders*. Below are slightly simplified versions:

```
zcb maturity principal asset =
 when (time == maturity) (scale (O.pure principal) (one asset))
```

Here we've just wrapped our expression from the previous section in a function `zcb`, that we can reuse to build a `fixed`-rate bond:

```
fixed : Decimal -> Decimal -> a -> [Date] -> Claim a
fixed principal coupon asset [] = zero
fixed principal coupon asset [maturity] =
  zcb maturity coupon asset `and` zcb maturity principal asset
fixed principal coupon asset (t :: ts) =
  zcb t coupon asset `and` fixed principal coupon asset ts
```

We define the fixed rate bond by induction, iterating over a list of dates `[t]`, and producing multiple zero-coupon bonds `zcb` combined together with `and`:

> The first definition covers the trivial case where we pass an empty list of dates.
> The second definition handles the base case, at maturity: we create both a coupon (interest) payment, and the principal payment.
> The third definition is the induction step; it peels the first element off the list, and calls itself recursively on the tail of the list, until it reaches the base case at maturity.

This re-use of code is prevalent throughout the library. It's great as it mirrors how instruments are defined in the industry. Let's look at yet another example, a fixed vs floating USD/EUR swap.

```
type Ccy = Text
usdVsEur : [Date] -> Claim Ccy
usdVsEur =
   fixed 100.0 0.1 "USD" `swap` floating (spot "EURUSD" * pure 100.0) (spot
↪"EURUSD") "EUR"
```

We define it in terms of its two legs, `fixed` and `floating`, which themselves are functions. We use `swap` in infix form, and partially apply it - it takes a final `[Date]` argument which we omit, hence the resulting signature `[Date] -> Claim Ccy`.

As you can see, not only is this approach highly composable, but it also mirrors the way derivative instruments are modelled in finance.

Another major advantage of this approach is its extensibility. Unlike a traditional approach, where we might in an object-oriented language represent different instruments as classes, in the cashflow approach, we do not need to enumerate possible asset classes/instruments *a priori*. This is especially relevant in a distributed setting, where parties must execute the same code, i.e., have the same `*.dar`s on their ledger to interact. In other words, party A can issue a new instrument, or even write a new combinator function that is in a private `*.dar`, while being able to trade with party B, who has no knowledge of this new `*.dar`.

Concerning Type Parameters

The curious reader may have noticed that the signature we gave for `data Claim` is not quite what is in the library, where we have `data Claim t x a o`. In our examples, we have specialised this to `type Claim' t x a o = Claim Date Decimal a a`. Parameterising these variables allows us to reason about `Asset`s and `Observation`s that appear in`Claim`s as function-like objects. The main use of this is to create claims with 'placeholders' for actual parameters, that can later be 'filled in' by mapping over them (`mapParams`).

The Time Parameter

`t` is used to represent the first input argument to an `Observation`, and above we used `Date` for this purpose. One reason this has been left parametrised is to be able to distinguish different calendar and day count conventions at the type level. This is quite a technical topic, but it suffices to know that for financial calculations, interest is not always accrued the same way, nor is settlement possible every day, as this depends on local jurisdictions or market conventions. Having different types makes this explicit at the instrument level.

Another use for this is expressing time as an ordinal values, representing e.g. days from issue. Such a `Claim` can be used repeatedly to represent instruments issued at different dates, but with the same durations. For example, consider a series of listed futures or options which are issued with quarterly/monthly maturities - their duration is about the same, but they are issued on different dates.

The Asset Parameter

`a`, as we already explained, is the type used to represent assets in your application. Keeping this generic means the library can be used with any asset representation. For example, you could use one of the instrument implementations in Daml Finance, but are not forced to do so.

The Observation Parameter

`o` is the type used to represent `Observation`s, which are time-dependent quantities that can be observed at any given time (such as the EURUSD exchange rate in the example above).

The Value Parameter

`x` is the 'output' type of an `Observation`, but it can also serve as input when defining a constant observation using, e.g., `Observation.pure 10.08`.

Lifecycling

So far we've learned how to model arbitrary financial instruments by representing them as trees of cashflows. We've seen that these trees can be constructed using the type constructors of `data Claim`, and that they can be factored into more complex building blocks using function composition. But now that we have these trees, what can we do with them?

The original paper [Cit1] focuses on using these trees for valuing the instruments they represent, i.e., finding the 'fair price' that one should pay for these cashflows. Instead, we'll focus here on a different use case: the lifecycling (aka safekeeping, processing corporate actions) of these instruments.

Let's go back to our fixed-rate bond example, above. We want to process the coupon payments. There is a function in the *Lifecycle module* for doing exactly this:

```
type C t a o = Claim t Decimal a o

-- | Used to specify pending payments.
data Pending t a = Pending
  with
    t : t
      -- ^ Payment time.
    amount : Decimal
      -- ^ Amount of asset to be paid.
    asset : a
      -- ^ Asset in which the payment is denominated.
  deriving (Eq, Show)

-- | Returned from a `lifecycle` operation.
data Result t a o = Result
  with
    pending : [Pending t a]
      -- ^ Payments requiring settlement.
    remaining : C t a o
      -- ^ The tree after lifecycled branches have been pruned.
  deriving (Eq, Show)

-- | Collect claims falling due into a list, and return the tree with those nodes
↪pruned.
-- `m` will typically be `Update`. It is parametrised so it can be run in a
↪`Script`. The first
-- argument is used to lookup the value of any `Observables`. Returns the pruned
↪tree + pending
-- settlements up to the provided market time.
lifecycle : (Ord t, Eq a, CanAbort m)
  => (o -> t -> m Decimal)
  -- ^ Function to evaluate observables.
  -> C t a o
  -- ^ The input claim.
  -> t
  -- ^ The input claim's acquisition time.
  -> t
  -- ^ The current market time. This is the time up to which observations are
↪known.
  -> m (Result t a o)
```

This may look daunting, but let's look at an example in ContingentClaims/Test/FinancialCon-

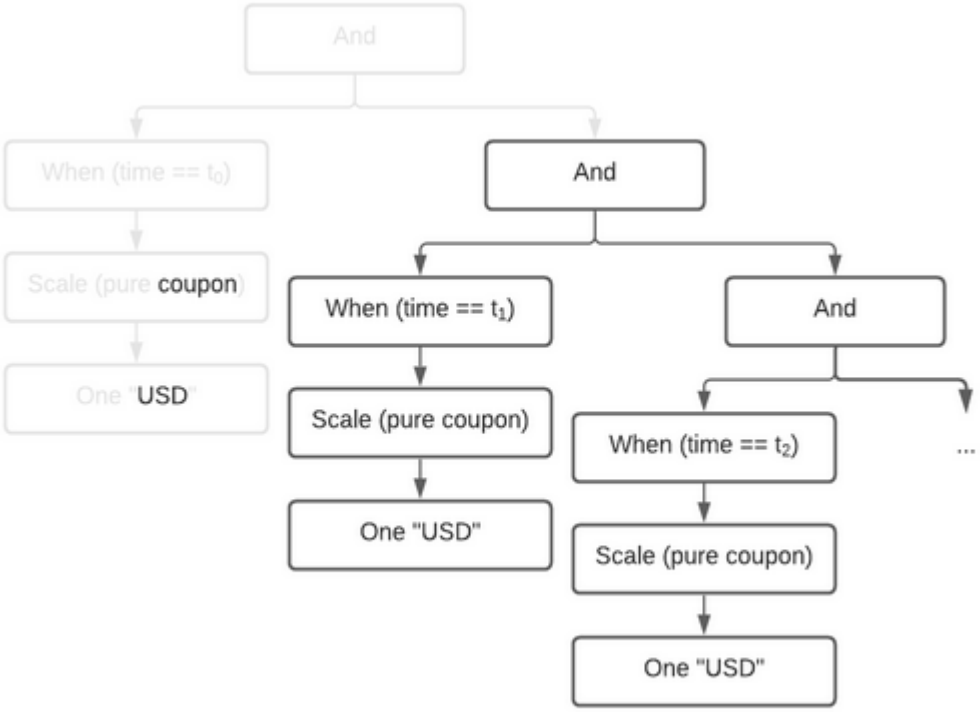
tract.daml to see this in action:

```
        do
          t <- toDateUTC <$> getTime
          let
            getSpotRate isin t = do
              (_, Quote{close}) <- fetchByKey (isin, t, bearer)
              pure close
          lifecycleResult <- Lifecycle.lifecycle getSpotRate claims
↪acquisitionTime t
```

The first argument to lifecycle, `getSpotRate`, is a function taking an ISIN (security) code, and today's date. All this does is fetch a contract from the ledger that is keyed by these two values, and extract the price of the security.

The last two arguments are simply the claims we wish to process, and today's date, evaluated using `getTime`.

The return value, `lifecycleResult`, will contain both the remaining tree after lifecycling, and any assets that need to be settled. In our running bond example, we would extract the `coupon` from the first payment, and return it, along with the rest of the tree, after that branch has been pruned (depicted greyed-out below):



You may wonder why we've separated the settlement procedure from the lifecycling function. The reason is that we can't assume that settlement will happen on-chain; if it does, that is great, as we can embed this call into a template choice, and lifecycle and settle atomically. However, in the case where settlement must happen off-chain, there's no way to to do this in one step. This design supports both choices.

Pricing (Experimental)

This is an *experimental* feature. Expect breaking changes.

The *ContigentClaims.Valuation.Stochastic* module can be used for valuation. There is a `fapf` function which is used to derive a *fundamental asset pricing formula* for an arbitrary `Claim` tree. The resulting AST is represented by `Expr`, but can be rendered as XML/MathML with the provided `MathML.presentation` function, for display in a web browser. See the `Test/Pricing` module for examples. Here is a sample rendering of a margrabe option:

```
<math display="block"><msub><mi>USD</mi><mi>t</mi></msub><mo>□</mo><mo>□</mo><mo>□
↪</mo><mrow><mo fence="true">[</mo><mrow><mo fence="true">(</mo><msub><mo>I</mo>
↪<mrow><msub><mi>AMZN</mi><mi>T</mi></msub><mo>-</mo><msub><mi>APPL</mi><mi>T</
↪mi></msub><mo>≤</mo><mn>0.0</mn></mrow></msub><mo>□</mo><mrow><mo fence="true">(
↪</mo><msub><mi>AMZN</mi><mi>T</mi></msub><mo>-</mo><msub><mi>APPL</mi><mi>T</mi>
↪</msub><mo fence="true">)</mo></mrow><mo>+</mo><msub><mo>I</mo><mrow><mn>0.0</
↪mn><mo>≤</mo><msub><mi>AMZN</mi><mi>T</mi></msub><mo>-</mo><msub><mi>APPL</mi>
↪<mi>T</mi></msub></mrow></msub><mo>×</mo><mn>0.0</mn><mo fence="true">)</mo></
↪mrow><mo>□</mo><msup><mrow><msub><mi>USD</mi><mi>T</mi></msub></mrow><mrow><mo>-
↪</mo><mn>1.0</mn></mrow></msup><mo>|</mo><msub><mo mathvariant="script">F</mo>
↪<mi>t</mi></msub><mo fence="true">]</mo></mrow></math>
```

You can cut-and-paste this into a web page in 'developer mode' in any modern browser.

References

The papers can be downloaded from Microsoft Research.

1.20 Packages

This section explains the different packages in Daml Finance. It describes the modules contained in each package and points to resources for learning how to use the packages (either a tutorial or a test that describes how to use the package step-by-step).

1.20.1 Interface Packages

This section lists the interface packages contained within Daml Finance:

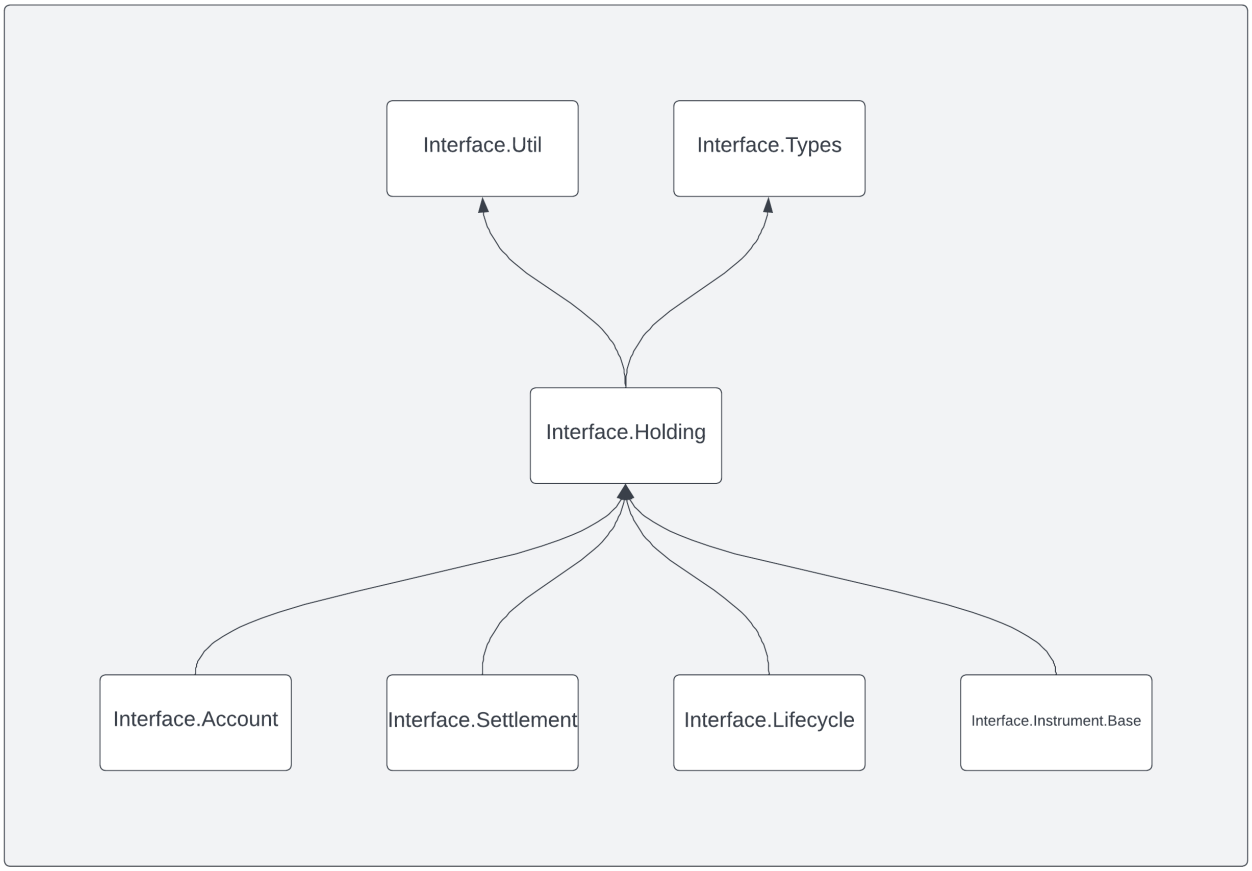
1.20.1.1 Daml.Finance.Interface.Holding

This package contains the *interface* and utility functions for holdings. It has the following modules:

Factory: Interface for a holding factory used to create (credit) and archive (debit) holdings
Base: Interface for a base holding which includes locking capabilities
Fungible: Interface for a fungible holding which allows splitting and merging
Transferable: Interface for a transferable holding, i.e., where ownership can be transferred to other parties
Util: Utility functions related to holdings, e.g., getting the amount or the instrument of a holding

The *Asset Model* page explains the relationship between instruments, holdings, and accounts. Check out the *Transfer tutorial* for a description on how to create a holding on an instrument and how to transfer it between accounts.

The following diagram shows the incoming and outgoing dependencies for this package:



Changelog

Daml.Finance.Interface.Holding - Changelog

Version 2.0.0

Update of SDK version and dependencies
Remove implementation of *Remove* choice from factory templates
Removed unnecessary *ArchiveFungible* choice
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure*, *asBase*, and *asTransferable* methods were removed)
Fix to signature of *disclose* (removed the *actor* argument)

1.20.1.2 Daml.Finance.Interface.Account

This package contains the *interface* and utility functions for accounts. It has the following modules:

Factory: Interface that allows implementing templates to create and remove accounts
Account: Interface which represents an established relationship between a custodian and an owner. It specifies parties controlling incoming and outgoing transfers, and allows for crediting and debiting holdings
Util: Utility functions related to accounts, e.g., getting the custodian or the owner of an account

Changelog

Daml.Finance.Interface.Account - Changelog

Version 2.0.0

Update of SDK version and dependencies
Remove type synonym for *AccountKey*
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure* method was removed)

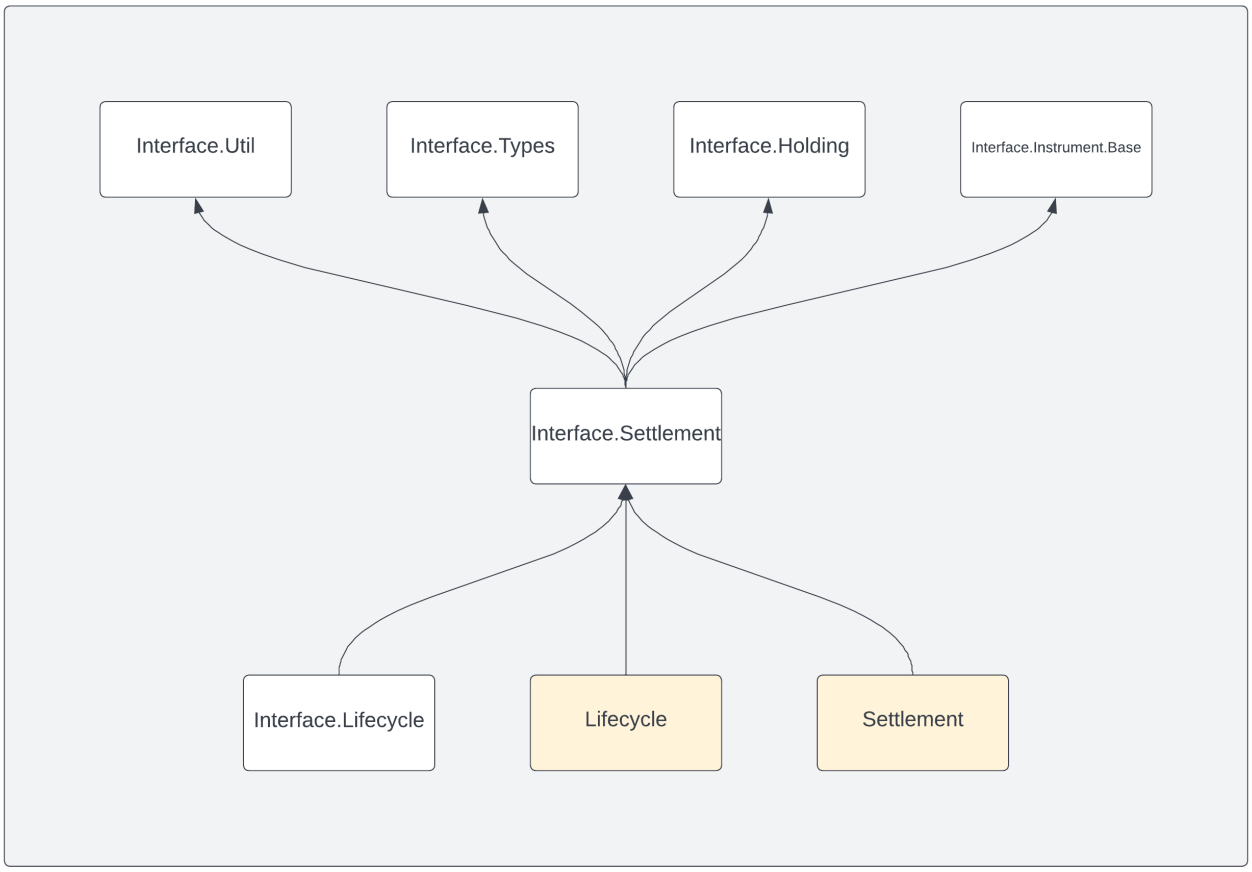
1.20.1.3 Daml.Finance.Interface.Settlement

This package contains the *interface* for settlement. It has the following modules:

RouteProvider: Interface for providing a discovery mechanism for settlement routes
Instruction: Interface for providing a single instruction to transfer an asset at a custodian
Batch: Interface for atomically executing instructions, i.e., settling *Transferable*s
Factory: Interface used to generate a batch and associated instructions
Types: Types required in the settlement process, e.g., *Step*, *RoutedStep*, *Allocation*, and *Approval*

The *Settlement* page contains an overview of the settlement process and explains the relationship between instructions and batches. Check out the *Settlement tutorial* for a description on how to use the settlement workflow in practice.

The following diagram shows the incoming and outgoing dependencies for this package:



Changelog

Daml.Finance.Interface.Settlement - Changelog

Version 2.0.0

Update of SDK version and dependencies
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure* was removed)

1.20.1.4 Daml.Finance.Interface.Lifecycle

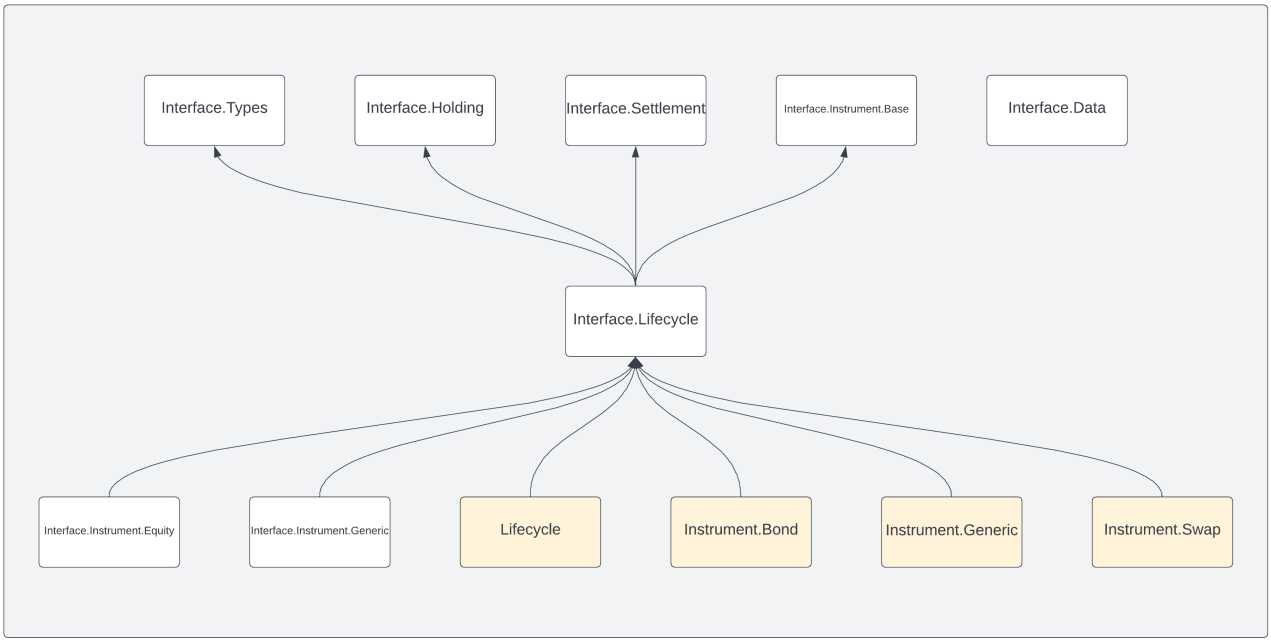
This package contains the *interface* for lifecycle related processes. It contains the following modules:

Effect: Interface for contracts exposing effects of lifecycling processes, e.g., the payment resulting from a bond coupon
Election: Interface to allow for elections to be made on claim based instruments
Election.Factory: Factory interface to instantiate elections on claim based instruments
Event: Interface for a lifecycle event. An event is any contract that triggers the processing of a lifecycle rule. Events can be, e.g., dividend announcements or simply the passing of time.
Event.Distribution: Event interface for the distribution of units of an instrument for each unit of a target instrument (e.g. share or cash dividends)

Event.Replacement: Event interface for the replacement of units of an instrument with a basket of other instruments (e.g. stock merger)
Event.Time: Event interface for events that signal the passing of (business) time
Rule.Lifecycle: Interface implemented by rules that lifecycle and evolve instruments
Rule.Claim: Interface for contracts that allow holders to claim an `Effect` and generate settlement instructions
Observable.NumericObservable: Interface to observe time-dependent numerical values (e.g. a stock price or an interest rate fixing)
Observable.TimeObservable: Interface implemented by templates exposing time information

The *Lifecycling* page contains an overview of the lifecycle process and explains the relationship between events, lifecycle rules and effects. Check out the *Lifecycling tutorial* for a description on how lifecycling works in practice. There is also the tutorial *How to implement a Contingent Claims-based instrument*, which describes how claims are defined, how to use a `NumericObservable`, and how the `Lifecycle` interface is implemented for bonds.

The following diagram shows the incoming and outgoing dependencies for this package:



Changelog

Daml.Finance.Interface.Lifecycle - Changelog

Version 2.0.0

Update of SDK version and dependencies
Remove implementation of *Remove* choice from factory interfaces
Move the *Election* module from the *Generic* to the *Lifecycle* package
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure* and *asEvent* methods were removed)

1.20.1.5 Daml.Finance.Interface.Instrument.Base

This package contains the *root interface* for all instruments. It contains the following modules:

Instrument: Root instrument abstraction containing basic properties that every instrument exhibits

Changelog

Daml.Finance.Interface.Instrument.Base - Changelog

Version 2.0.0

Update of SDK version and dependencies
Removed type synonym for *InstrumentKey*
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure* method was removed)

1.20.1.6 Daml.Finance.Interface.Claims

This package contains the *interface* for Contingent Claims based instruments. It contains the following modules:

Claim: Interface implemented by templates that can be represented as a set of contingent claims
Dynamic.Instrument: Interface implemented by instruments that create Contingent Claims trees on-the-fly (i.e. the tree is not stored on disk as part of a contract, but created and processed in-memory)
Types: Types related to claims and what is required to represent claims (e.g. Deliverable and Observable)

Changelog

Daml.Finance.Interface.Claims - Changelog

Version 2.0.0

Update of SDK version and dependencies
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asBaseInstrument* method was removed)

1.20.1.7 Daml.Finance.Interface.Data

This package contains the *interface* for inspecting and working with observables, which are used in the context of lifecycling. It contains the following modules:

Numeric.Observation.Factory: Interface for a factory used to create, remove and view a `Numeric.Observation`
Numeric.Observation: Interface for a time-dependent `Numeric.Observation`, where the values are explicitly stored on-ledger
Reference.HolidayCalendar.Factory: Interface for a factory used to create, remove and view a `HolidayCalendar`
Reference.HolidayCalendar: Interface for contracts storing holiday calendar data on the ledger
Reference.Time: Interface for contracts that control business time, providing choices to advance or rewind time

Changelog

Daml.Finance.Interface.Data - Changelog

Version 3.0.0

Update of SDK version and dependencies
Remove implementation of *Remove* choice from factory templates
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure*, *asNumericObservable*, and *asTimeObservable* methods were removed)

Version 2.0.0

Changed the signature of *advance* and *rewind* in the *Reference.Time* interface
Dependencies update

1.20.1.8 Daml.Finance.Interface.Types.Common

This package contains common type definitions. They are defined in the following modules:

Types: Various types related to keys, observers, parties, identifiers and quantities, which are commonly used in several packages

Changelog

Daml.Finance.Interface.Types.Common - Changelog

Version 1.0.1

Update of SDK version and dependencies

1.20.1.9 Daml.Finance.Interface.Types.Date

This package contains types related to dates. They are defined in the following modules:

Date.Calendar: Types for holiday calendar data and how to adjust non-business days
Date.Classes: Type class that specifies what can be converted to UTC time
Date.DayCount: Type to specify the conventions used to calculate day count fractions
Date.RollConvention: Types to define date periods and how to roll dates
Date.Schedule: Types to define date schedules

Changelog

Daml.Finance.Interface.Types.Date - Changelog

Version 2.0.1

Update of SDK version and dependencies

Version 2.0.0

Introduce *ScheduleFrequency* data type
Added *NoRollConvention* to *RollConventionEnum*. It applies to *D* and *W* periods
PeriodicSchedule.frequency is of type *ScheduleFrequency* instead of *Frequency*

1.20.1.10 Daml.Finance.Interface.Util

This package contains the *interface* for the disclosure of contracts and some commonly used utility functions. They are defined in these modules:

Disclosure: An interface for managing the visibility of contracts for non-authorizing parties
Common: Different utility functions related to interfaces and assertions

Changelog

Daml.Finance.Interface.Util - Changelog

Version 2.0.0

Update of SDK version and dependencies
Remove *mapWithIndex* utility function
Remove the *HasImplementation* type class definition

1.20.1.11 ContingentClaims.Core

This package contains the *interface* to represent *Contingent Claims* trees. It contains data types and utility functions to process such trees. It also contains builder functions to facilitate the creation of trees using composition. The following modules are included:

Builders: Builder functions to compose trees from smaller building blocks
Internal.Claim: Internal data types to represent tree nodes
Claim: Smart constructors for the types defined in *Internal.Claim*
Observation: Data types to represent observations in trees
Util.Recursion: Utility functions to facilitate recursive traversal of trees

Changelog

ContingentClaims.Core - Changelog

Version 2.0.0

Update of SDK version and dependencies
Add *orList* and *andList* smart constructors
Add *ObserveAt* observation builder, used to explicitly state when an *Observation* should be observed
Refactor *or* and *anytime* smart constructors to identify electable sub-trees by a textual tag

1.20.1.12 Daml.Finance.Interface.Instrument.Bond

This package contains the *interface* definitions for bond instruments. It contains the following modules:

Callable.Instrument: Instrument interface for callable bonds
Callable.Factory: Factory interface to instantiate callable bond instruments
Callable.Types: Type definitions to support callable bond instruments
FixedRate.Instrument: Instrument interface for fixed-rate bonds
FixedRate.Factory: Factory interface to instantiate fixed-rate bond instruments
FixedRate.Types: Type definitions to support fixed-rate bond instruments
FloatingRate.Instrument: Instrument interface for floating-rate bonds
FloatingRate.Factory: Factory interface to instantiate floating-rate bond instruments
FloatingRate.Types: Type definitions to support floating-rate bond instruments
InflationLinked.Instrument: Instrument interface for inflation-linked bonds
InflationLinked.Factory: Factory interface to instantiate inflation-linked bond instruments
InflationLinked.Types: Type definitions to support inflation-linked bond instruments
ZeroCoupon.Instrument: Instrument interface for zero-coupon bonds
ZeroCoupon.Factory: Factory interface to instantiate zero-coupon bond instruments
ZeroCoupon.Types: Type definitions to support zero-coupon bond instruments
Types: Type definitions common to different types of bonds

Changelog

Daml.Finance.Interface.Instrument.Bond - Changelog

Version 1.0.0

Update of SDK version and dependencies
The *Create* choice on the instrument factories returns the corresponding interface (rather than the base instrument interface)
Add *GetView* choice to all instrument interfaces
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure* and *asBaseInstrument* methods were removed)
Introduce a new callable bond instrument
Add a *notional* field to all instruments

Version 0.2.1

Dependencies update

1.20.1.13 Daml.Finance.Interface.Instrument.Equity

This package contains the *interface* definitions for equity instruments. It contains the following modules:

Instrument: Instrument interface for equities. It supports lifecycling events through the `DeclareDistribution`, `DeclareReplacement` and `DeclareStockSplit` choices.
Factory: Factory interface to instantiate equities

Changelog

Daml.Finance.Interface.Instrument.Equity - Changelog

Version 0.3.0

Update of SDK version and dependencies
The *Create* choice on the instrument factory returns the corresponding interface (rather than the base instrument interface)
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure* and *asBaseInstrument* methods were removed)
Rename *DeclareDividend* to *DeclareDistribution*

1.20.1.14 Daml.Finance.Interface.Instrument.Generic

This package contains the *interface* definitions for generic, contingent-claims-based instruments. It contains the following modules:

Instrument: Instrument interface for generic instruments
Factory: Factory interface to instantiate generic instruments

Changelog

Daml.Finance.Interface.Instrument.Generic - Changelog

Version 2.0.0

Update of SDK version and dependencies
The *Create* choice on the instrument factory returns the corresponding interface (rather than the base instrument interface)
Move the *Election* module to the *Lifecycle* package. Also, refactor the *Election* to identify the elected sub-tree by a textual tag rather than the actual sub-tree
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure*, *asBaseInstrument*, and *asClaim* methods were removed)

1.20.1.15 Daml.Finance.Interface.Instrument.Option

This package contains the *interface* definitions for various option instruments. It contains the following modules:

BarrierEuropeanCash.Instrument: Instrument interface for barrier options
BarrierEuropeanCash.Factory: Factory interface to instantiate barrier options
BarrierEuropeanCash.Types: Type definitions to support barrier options
Dividend.Instrument: Instrument interface for dividend options
Dividend.Factory: Factory interface to instantiate dividend options
Dividend.Types: Type definitions to support dividend options
Dividend.Election.Factory: Factory interface to instantiate elections for dividend options
EuropeanCash.Instrument: Instrument interface for cash-settled European options
EuropeanCash.Factory: Factory interface to instantiate cash-settled European options
EuropeanCash.Types: Type definitions to support cash-settled European options
EuropeanPhysical.Instrument: Instrument interface for physically settled European options
EuropeanPhysical.Factory: Factory interface to instantiate physically settled European options
EuropeanPhysical.Types: Type definitions to support physically settled European options
Types: Type definitions common to several instruments in this package

Changelog

Daml.Finance.Interface.Instrument.Option - Changelog

Version 0.2.0

Update of SDK version and dependencies
The *Create* choice on the instrument factories returns the corresponding interface (rather than the base instrument interface)
Add instruments physically-settled European options, dividend options, barrier options
Renamed cash-settled European options to *EuropeanCash*
Added *GetView* choice to all instrument interfaces
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure*, *asBaseInstrument*, and *asEvent* methods were removed)

1.20.1.16 Daml.Finance.Interface.Instrument.Swap

This package contains the *interface* definitions for various swap instruments. It contains the following modules:

Asset.Instrument: Instrument interface for asset swaps
Asset.Factory: Factory interface to instantiate asset swaps
Asset.Types: Type definitions to support asset swaps
CreditDefault.Instrument: Instrument interface for credit default swaps
CreditDefault.Factory: Factory interface to instantiate credit default swaps
CreditDefault.Types: Type definitions to support credit default swaps
Currency.Instrument: Instrument interface for currency swaps
Currency.Factory: Factory interface to instantiate currency swaps
Currency.Types: Type definitions to support currency swaps
ForeignExchange.Instrument: Instrument interface for foreign exchange swaps
ForeignExchange.Factory: Factory interface to instantiate foreign exchange swaps
ForeignExchange.Types: Type definitions to support foreign exchange swaps
Fpml.Instrument: Instrument interface for FpML swaps
Fpml.Factory: Factory interface to instantiate FpML swaps
Fpml.FpmlTypes: Specific FpML types used to support FpML swaps
Fpml.Types: Type definitions to support FpML swaps
InterestRate.Instrument: Instrument interface for interest rate swaps
InterestRate.Factory: Factory interface to instantiate interest rate swaps
InterestRate.Types: Type definitions to support interest rate swaps

Changelog

Daml.Finance.Interface.Instrument.Swap - Changelog

Version 0.3.0

Update of SDK version and dependencies
The *Create* choice on the instrument factories returns the corresponding interface (rather than the base instrument interface)

Added *GetView* choice to all instrument interfaces
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure* and *asBaseInstrument* methods were removed)

Version 0.2.1

Updates to data types related to interest rate compounding and payment lag
Updates to data types related to Term period

1.20.1.17 Daml.Finance.Interface.Instrument.Token

This package contains the *interface* definitions for simple token instruments which do not define any lifecycling logic. It contains the following modules:

Instrument: Instrument interface for simple tokens
Factory: Factory interface to instantiate simple tokens
Types: Type definitions to support simple tokens

Check out the *Transfer tutorial* for an example on how to create a simple token instrument and use it for a transfer.

Changelog

Daml.Finance.Interface.Instrument.Token - Changelog

Version 2.0.0

Update of SDK version and dependencies
The *Create* choice on the instrument factory returns the corresponding interface (rather than the base instrument interface)
Make use of the *requires* keyword to enforce the interface hierarchy (in the particular *asDisclosure* and *asBaseInstrument* implementations were removed)

1.20.2 Implementation Packages

This section lists the implementation packages contained within Daml Finance:

1.20.2.1 Daml.Finance.Holding

This package contains the *implementation* of holdings, including utility functions. It has the following modules:

Fungible: Implementation of a fungible holding, including split and merge functionality
NonFungible: Implementation of a non-fungible holding, which cannot be split or merged
NonTransferable: Implementation of a non-transferable holding
Util: Utility functions related to holdings, e.g., to transfer or lock/release a holding

The *Asset Model* page explains the relationship between instruments, holdings, and accounts. Also, check out the *Transfer tutorial* for a description of how to create a holding on an instrument and transfer it between accounts.

Changelog

Daml.Finance.Holding - Changelog

Version 2.0.0

Update of SDK version and dependencies
Remove implementation of *Remove* choice from factory templates
Added default *splitImpl* and *mergeImpl* for *Fungible* to *Util.daml*
Generalized the *acquireImpl* and *releaseImpl* to not rely on an attribute called lock
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure*, *asBase*, and *asTransferable* implementations were removed)
The *Transfer* choice of the *Transferable* interface now includes the new owner as a choice observer
Implementation of *Lockable* does not allow an empty *lockers* set

Version 1.0.2

Dependencies update

Version 1.0.1

Fix bug in the implementation of *Fungible.Merge*
Improve error message when acquiring a lock

1.20.2.2 Daml.Finance.Account

This package contains the *implementation* of accounts. It has the following module:

Account: Implementation of an account, i.e., a relationship between a custodian and an asset owner, referenced by holdings. It also provides an implementation of a factory from which you can create and remove accounts. Upon creation of an account, it allows you to specify controlling parties for incoming / outgoing transfers.

Changelog

Daml.Finance.Account - Changelog

Version 2.0.0

Update of SDK version and dependencies
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure* implementation was removed)

Use *ensure* to ensure that the set of outgoing controllers is non-empty

Version 1.0.1

Dependencies update

1.20.2.3 Daml.Finance.Settlement

This package contains the *implementation* of the components used for settlement. It has the following modules:

RouteProvider.SingleCustodian: Used to generate a single *RoutedStep* from a *Step* using a single custodian
RouteProvider.IntermediatedStatic: Used to generate a route, i.e., *RoutedSteps*, for each settlement *Step*
Instruction: Used to settle a single *RoutedStep*, i.e., a *Step* at a custodian.
Batch: Allows you to atomically settle a set of settlement *Instructions*
Factory: Used to create a set of settlement *Instructions*, and a *Batch* to atomically settle them
Hierarchy: Data type that describes a hierarchical account structure between multiple parties

The *Settlement* page contains an overview of the settlement process and explains the relationship between `Instruction` and `Batch`. Also, check out the *Settlement tutorial* for a description on how to implement the settlement workflow in practice.

Changelog

Daml.Finance.Settlement - Changelog

Version 2.0.0

Update of SDK version and dependencies
In the settlement *Factory*, the id values used for the *Instruction`s were modified to accurately reflect their order within the `Batch*.
In the *Batch*, the order of the *settledCids* were changed to match the initial order of the instructions in the batch.
Bug fix: replace *groupOn* by *sortAndGroupOn* in the *Instruction* and *IntermediatedStatic* templates
Lock pledged holding when allocating to an *Instruction*: the pledged holding is locked to the instruction's requestors and the outgoing controllers of the sending account
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure* implementation was removed)
When an *Allocation* (resp. *Approval*) takes place, a check has been added to ensure that either the *sender* or the *custodian* (resp. the *receiver* or the *custodian*) is among the choice authorizers
When reallocation (resp. re-approval) occurs, it is required that the *signedSenders* (resp. *signedReceivers*) of the *Instruction* are part of the authorizing set
Add additional checks to the pass-through allocation/approval process. Specifically, verify that the specified pass-through *Instruction* is actually part of the *Batch*. These checks detect settlement failures during the allocation/approval stage rather than waiting until settlement occurs.
Removed the *key* from the *Batch* implementation

Version 1.0.2

Dependencies update

Version 1.0.1

Additional sanity checks added to *Instruction*

1.20.2.4 Daml.Finance.Lifecycle

This package contains the *implementation* of lifecycle related processes. It contains the following modules:

Effect: A contract encoding the *consequences of a lifecycle event* for one unit of the target instrument
Election: Implementation of elections (e.g. the exercise of an option) for claim based instruments
ElectionEffect: A contract encoding the *consequences of an election* for one unit of the target instrument
Rule.Claim: Rule contract that allows an actor to process/claim effects, returning settlement instructions
Rule.Distribution: Rule contract that defines the distribution of units of an instrument for each unit of a target instrument (e.g. share or cash dividends)
Rule.Replacement: Rule contract that defines the replacement of units of an instrument with a basket of other instruments (e.g. stock merger)
Rule.Util: Utility functions to net, split and merge pending payments
Event.Distribution: Event contract for the distribution of units of an instrument for each unit of a target instrument (e.g. share or cash dividends)
Event.Replacement: Event contract for the replacement of units of an instrument with a basket of other instruments (e.g. stock merger)

Check out the *Lifecycling tutorial* for a description on how lifecycling works in practice, including how to *Claim* an *Effect*. There is also the tutorial *How to implement a Contingent Claims-based instrument*, which describes how to create an *Effect*. For a description of `Distribution` and `Replacement`, check out the Instrument/Equity/Test folder. It demonstrates how to create and lifecycle a cash dividend, and how to handle corporate actions like mergers and stock splits.

Changelog

Daml.Finance.Lifecycle - Changelog

Version 2.0.0

Update of SDK version and dependencies
Remove implementation of *Remove* choice from factory templates
Move the *Election* module from the *Generic* to the *Lifecycle* package
Election and *ElectionEffect* implement the *Disclosure* interface
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure* and *asEvent* implementations were removed)

The *Distribution* and *Replacement* lifecycle rules check that the target and procued instruments are active

Version 1.0.1

Dependencies update

1.20.2.5 Daml.Finance.Data

This package implements templates containing reference data. It includes the following modules:

Numeric.Observation: An implementation of an observation that explicitly stores time-dependent numerical values on the ledger. It can be used to, e.g., store equity or rate fixings.
Reference.HolidayCalendar: A holiday calendar of an entity (typically an exchange or a currency)
Time.DateClock.Types: A date type which can be converted to time, and time-related utility functions
Time.DateClock: A contract specifying what is the current local date. It is used to inject date information in lifecycle processing rules
Time.DateClockUpdate: A contract representing passing of (market) time that can be used to trigger contractual, time-based cashflows, like interest payments on a bond. It is, for example, used to drive the evolution and lifecycling of *Contingent Claims*-based instruments.
Time.LedgerTime: A time observable which uses ledger time

Changelog

Daml.Finance.Data - Changelog

Version 2.0.0

Update of SDK version and dependencies
Remove implementation of *Remove* choice from factory templates
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure*, *asNumericObservable*, *asTimeObservable*, and *asEvent* implementations were removed)
Removed *key* from *DateClock*.

Version 1.0.1

Fixed a bug in the *DateClock* implementation to avoid key violations
Dependencies update

1.20.2.6 Daml.Finance.Claims

This package contains utility functions that facilitate building and working with *Contingent Claims* based instruments. It includes the following modules:

Lifecycle.Rule: Rule to process a lifecycle event for instruments that build a *Contingent Claims tree dynamically*.
Util: Contains utility functions for claims, e.g., checking the content of a claim and converting the claim time
Util.Lifecycle: Defines different types of events and how to lifecycle them
Util.Builders: Utility functions related to creating *Contingent Claims*, e.g. for bonds/swaps
Util.Date: Utility functions related to dates and schedule periods, which are used to define claims.

Changelog

Daml.Finance.Claims - Changelog

Version 2.0.0

Update of SDK version and dependencies
The lifecycle rule supports a combination of elections and time-based events
Improve tagging of new instrument versions in the lifecycle rule
A new instrument version is created and returned by the *Evolve* choice also when the instrument expires

Version 1.0.1

Dependencies update

1.20.2.7 Daml.Finance.Util

This package mainly contains utility functions related to dates, lists, maps, and disclosure. They are defined in the following modules:

Date.Calendar: Functions regarding dates and holiday calendars (business vs non-business days)
Date.DayCount: Functions to calculate day count fractions according to different conventions
Date.RollConvention: Functions to calculate date periods including rolling dates
Date.Schedule: Functions to calculate a periodic schedule, including both adjusted and unadjusted dates
Common: Various functions related to lists and maps, which are used in several packages
Disclosure: Utility functions related to disclosure, e.g., to add or remove observers

Changelog

Daml.Finance.Util - Changelog

Version 3.0.0

Update of SDK version and dependencies
Remove the *groupBy* utility function

Version 2.0.0

calcPeriodDcf and *calcPeriodDcfActActISMA* take a *ScheduleFrequency* instead of a *Frequency*
Dependencies update

1.20.2.8 ContingentClaims.Lifecycle

This package contains the *implementation* of utility functions to lifecycle a *Contingent Claims* tree. The following modules are included:

Lifecycle: Functions to lifecycle a *Contingent Claims* tree
Util: Utility functions to query a *Contingent Claims* tree for certain properties

Changelog

ContingentClaims.Lifecycle - Changelog

Version 2.0.0

Update of SDK version and dependencies
Refactor *exercise* to identify the elected sub-tree by a textual tag rather than the actual sub-tree

1.20.2.9 ContingentClaims.Valuation

This package contains the *implementation* of utility functions to map a *Contingent Claims* tree into a mathematical representation to facilitate integration with pricing and risk frameworks. The following modules are included:

Stochastic: Utilities to map a *Contingent Claims* tree to a stochastic process representation
MathML: Typeclass definition to map an expression in the MathML presentation format

Changelog

ContingentClaims.Valuation - Changelog

Version 0.2.1

Update of SDK version and dependencies

1.20.2.10 Daml.Finance.Instrument.Bond

This package contains the *implementation* of different bond types, defined in the following modules:

Callable.Instrument: Instrument implementation for callable bonds
Callable.Factory: Factory implementation to instantiate callable bonds
FixedRate.Instrument: Instrument implementation for fixed-rate bonds
FixedRate.Factory: Factory implementation to instantiate fixed-rate bonds
FloatingRate.Instrument: Instrument implementation for floating-rate bonds
FloatingRate.Factory: Factory implementation to instantiate floating-rate bonds
InflationLinked.Instrument: Instrument implementation for inflation-linked bonds
InflationLinked.Factory: Factory implementation to instantiate inflation-linked bonds
ZeroCoupon.Instrument: Instrument implementation for zero-coupon bonds
ZeroCoupon.Factory: Factory implementation to instantiate zero-coupon bonds
Util: Bond-specific utility functions

Check out the page on *How to use the Bond Instrument packages* for a description of how to use these instruments in practice. There is also the tutorial *How to implement a Contingent Claims-based instrument*, which describes how the claims are defined and how the lifecycle interface is implemented for bonds.

Changelog

Daml.Finance.Instrument.Bond - Changelog

Version 1.0.0

Update of SDK version and dependencies
The *Create* choice on the instrument factories returns the corresponding interface (rather than the base instrument interface)
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure* and *asBaseInstrument* implementations were removed)
Introduce a new callable bond instrument
Add a *notional* field to all instruments

Version 0.2.1

Dependencies update

1.20.2.11 Daml.Finance.Instrument.Equity

This package contains the *implementation* of equity instruments, defined in the following modules:

Instrument: Instrument implementation for equities
Factory: Factory implementation to instantiate equities

For a detailed explanation of this package, check out the page on *How to use the Equity Instrument packages*. It demonstrates how to originate an equity instrument, how to create and lifecycle a cash dividend, and how to handle corporate actions like mergers and stock splits.

Changelog

Daml.Finance.Instrument.Equity - Changelog

Version 0.3.0

Update of SDK version and dependencies
The *Create* choice on the instrument factory returns the corresponding interface (rather than the base instrument interface)
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure* and *asBaseInstrument* implementations were removed)
Rename *DeclareDividend* to *DeclareDistribution*

Version 0.2.1

Dependencies update

1.20.2.12 Daml.Finance.Instrument.Generic

This package contains the *implementation* of generic, *Contingent Claims* based instruments, defined in the following modules:

Instrument: Instrument implementation for generic instruments
Factory: Factory implementation to instantiate generic instruments
Lifecycle.Rule: Rule to process a lifecycle event for generic instruments

Check out the page on *How to use the Generic Instrument packages* as well as the *Payoff modeling tutorial* to learn how to use this instrument in practice.

Changelog

Daml.Finance.Instrument.Generic - Changelog

Version 2.0.0

Update of SDK version and dependencies
The *Create* choice on the instrument factory returns the corresponding interface (rather than the base instrument interface)
Move the *Election* module to the *Lifecycle* package. Also, refactor the *Election* to identify the elected sub-tree by a textual tag rather than the actual sub-tree
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure*, *asBaseInstrument*, and *asClaim* implementations were removed)
A new instrument version is created and returned by the lifecycle rule choice also when the instrument expires

Version 1.0.1

Dependencies update

1.20.2.13 Daml.Finance.Instrument.Option

This package contains the *implementation* of different option types, defined in the following modules:

BarrierEuropeanCash.Instrument: Instrument implementation for barrier options
BarrierEuropeanCash.Factory: Factory implementation to instantiate barrier options
Dividend.Instrument: Instrument implementation for dividend options
Dividend.Factory: Factory implementation to instantiate dividend options
Dividend.Election: Factory implementation to create an Election for dividend options
EuropeanCash.Instrument: Instrument implementation for cash-settled European options
EuropeanCash.Factory: Factory implementation to instantiate cash-settled European options
EuropeanPhysical.Instrument: Instrument implementation for physically settled European options
EuropeanPhysical.Factory: Factory implementation to instantiate physically settled European options
Util: Utility functions for options

Check out the page on *How to use the Option Instrument packages* for a description of how to use the these instruments in practice.

Changelog

Daml.Finance.Instrument.Option - Changelog

Version 0.2.0

Update of SDK version and dependencies
The *Create* choice on the instrument factories returns the corresponding interface (rather than the base instrument interface)
Add instruments physically-settled European options, dividend options, barrier options
Renamed cash-settled European options to *EuropeanCash*
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure* and *asBaseInstrument* implementations were removed)

1.20.2.14 Daml.Finance.Instrument.Swap

This package contains the *implementation* of different swap types, defined in the following modules:

Asset.Instrument: Instrument implementation for asset swaps
Asset.Factory: Factory implementation to instantiate asset swaps
CreditDefault.Instrument: Instrument implementation for credit default swaps
CreditDefault.Factory: Factory implementation to instantiate credit default swaps
Currency.Instrument: Instrument implementation for currency swaps
Currency.Factory: Factory implementation to instantiate currency swaps
ForeignExchange.Instrument: Instrument implementation for foreign exchange swaps
ForeignExchange.Factory: Factory implementation to instantiate foreign exchange swaps
Fpml.Instrument: Instrument implementation for FpML swaps (FpML swap schema)
Fpml.Factory: Factory implementation to instantiate FpML swaps
Fpml.Util: Utility functions to support FpML swaps
InterestRate.Instrument: Instrument implementation for interest rate swaps
InterestRate.Factory: Factory implementation to instantiate interest rate swaps

Check out the page on *How to use the Swap Instrument packages* for a description of how to use these instruments in practice.

Changelog

Daml.Finance.Instrument.Swap - Changelog

Version 0.3.0

Update of SDK version and dependencies
The *Create* choice on the instrument factories returns the corresponding interface (rather than the base instrument interface)
Make use of the *requires* keyword to enforce the interface hierarchy (in particular the *asDisclosure* and *asBaseInstrument* implementations were removed)
FpmlSwap now accepts a non-zero rate fixing lag

Version 0.2.1

Implement interest rate compounding (several calculation periods per payment period)
Support a more generic way of specifying notional step schedules
Support specification of a payment lag
Efficient calculation of SOFR-like daily compounded reference rates
Implement arrears reset
Implement step-up coupon
Add support for initial stub period that starts before the issue date of the swap
Improve handling of principal exchange
Add support for Term period of a swap leg
Additional improvements required to make the official FpML trades 1..7 work as expected

1.20.2.15 Daml.Finance.Instrument.Token

This package contains the *implementation* of simple token instruments which do not define any life-cycling logic. It contains the following modules:

Instrument: Instrument implementation for simple tokens
Factory: Factory implementation to instantiate simple tokens

Check out the *Transfer tutorial* for an example on how to create a simple token instrument and use it for a transfer.

Changelog

Daml.Finance.Instrument.Token - Changelog

Version 2.0.0

Update of SDK version and dependencies
The *Create* choice on the instrument factory returns the corresponding interface (rather than the base instrument interface)
Make use of the *requires* keyword to enforce the interface hierarchy (in the particular *asDisclosure* and *asBaseInstrument* implementations were removed)

Version 1.0.1

Dependencies update

1.21 Tutorials

This section contains step-by-step implementation guides across different aspects of the Daml Finance library.

1.21.1 Getting Started tutorials

This section explains how some key concepts of Daml Finance work in practice. It combines a step by step description of different workflows with supporting Daml code.

The following tutorials are available:

Holdings: describes the core asset model used in Daml Finance.
Transfer: shows how to transfer ownership of a holding to another party.
Settlement: explains how to execute multiple asset movements atomically.
Lifecycling: describes how lifecycle rules and events can be used to evolve instruments over time.

Each tutorial builds on top of the previous ones, so they should ideally be followed in order.

1.21.1.1 Prerequisites

We expect the reader to be familiar with the basic building blocks of Daml. If that is not the case, a suitable introduction can be found here.

An understanding of *Daml Interfaces* is very helpful, as these are used extensively throughout the library. However, you should be able to follow along and grasp the fundamental concepts also without detailed knowledge on interfaces.

Finally, make sure that the *Daml SDK* is installed on your machine.

1.21.1.2 Download the code for the tutorials

Open a new terminal window and run:

```
daml new quickstart-finance --template quickstart-finance
```

This creates a new folder with contents from our template. Navigate to the folder and then run the following to download the required Daml Finance packages:

```
./get-dependencies.sh
```

or, if you are using Windows

```
./get-dependencies.bat
```

Finally, you can start Daml Studio to inspect the code and run the project's scripts:

```
daml studio
```

1.21.1.3 Structure of the Code and Dependencies

The project includes

four workflows defined in the `Workflows` folder
four Daml scripts defined in the `Scripts` folder

The `Workflows` encapsulate the core business logic of the application, whereas the `Scripts` are meant to be executed on a one-off basis.

As you can see from the import list, modules in the `Workflows` folder depend only on *interface* packages of Daml Finance (the packages that start with `Daml.Finance.Interface.*`).

This is important, as it decouples the user-defined business logic from the template implementations used in Daml Finance, which makes it easier to upgrade the application. The user-defined business logic in the `Workflows` will not need to be modified nor re-compiled to work with upgraded (ie., newer versions of) *implementation* packages.

On the other hand, modules in the `Scripts` folder depend on both the *interface* packages and the *implementation* packages (in this case, `Daml.Finance.Account`, `Daml.Finance.Holding`, and `Daml.Finance.Instrument.Token`). This is not problematic as scripts are meant to be run only once when the application is initialized.

1.21.1.4 Holdings

This tutorial introduces the core asset model of the library through a simple example. The purpose is to illustrate the concepts of *account*, *instrument*, and *holding*, as well as to show some useful patterns when working with Daml interfaces.

We are going to use the Daml Finance library to create tokenized cash on the ledger. This is done in three steps:

1. we first create accounts for Alice and Bob at the Bank
2. we then proceed to issue a cash instrument, representing tokenized dollars
3. we finally credit a cash holding to Alice's account

The holding contract represents the record of ownership on the ledger for Alice's tokenized cash.

Run the script

In order to show how this works in practice, let us explore the `Holding` script step-by-step.

Create Account, Holding, and Instrument Factories

The first instruction instantiates an account factory. This is a template that is used by a party (the Bank in this case) to create accounts as part of the `CreateAccount` workflow.

```
accountFactoryCid <- toInterfaceContractId @Account.F <$> submit bank do
  createCmd Account.Factory with
    provider = bank
    observers = empty
```

Notice how the `ContractId` is immediately converted to an interface upon creation: this is because our workflows, such as `CreateAccount`, do not have any knowledge of concrete template implementations.

Similarly, we define a *holding factory*, which is used within an account to create (`Credit`) holdings.

```
holdingFactoryCid <- toInterfaceContractId @Holding.F <$> submit bank do
  createCmd Fungible.Factory with
    provider = bank
    observers = fromList [("Settlers", S.fromList [alice, bob])]
```

This factory contract instantiates a specific implementation of holdings, which are defined in *Daml.Finance.Holding.Fungible* and are both *fungible*, as well as *transferable*.

Finally, we create a factory template which is used to instantiate *token instruments*.

```
tokenFactoryCid <- toInterfaceContractId @Token.F <$> submit bank do
  createCmd Token.Factory with
    provider = bank
    observers = empty
```

Open Alice's and Bob's Accounts

Once the factory templates are setup, we leverage the `CreateAccount` workflow to create accounts at the Bank for Alice and Bob.

The creation of an account needs to be authorized by both Alice and the Bank. Authorization is collected using a propose / accept pattern.

```
aliceRequestCid <- submit alice do
  createCmd CreateAccount.Request with
    owner = alice
    custodian = bank

aliceAccount <- submit bank do
  exerciseCmd aliceRequestCid CreateAccount.Accept with
    label = "Alice@Bank"
    description = "Account of Alice at Bank"
    accountFactoryCid = accountFactoryCid
    holdingFactoryCid = holdingFactoryCid
    observers = []
```

The Bank acts as the `custodian`, or account provider, whereas Alice is the account `owner`. Bob's account is created in a similar fashion.

Create the Cash Instrument

In order to credit Alice's account with some cash, we first create a cash instrument. An instrument is a representation of what it is that we are holding against the Bank. It can be as simple as just a textual label (like the *Token Instrument* used in this case) or it can include complex on-ledger lifecycling logic.

```
  let
    instrumentId = Id "USD"
    instrumentVersion = "0"
    instrumentKey = InstrumentKey with
      issuer = bank
      depository = bank
      id = instrumentId
      version = instrumentVersion
  now <- getTime

  submit bank do
    exerciseCmd tokenFactoryCid Token.Create with
      token = Token with
        instrument = instrumentKey
        description = "Instrument representing units of a generic token"
        validAsOf = now
      observers = empty
```

Notice how in this case the Bank acts both as the issuer and depository of the cash instrument. This means that we fully trust the Bank with any action concerning the instrument contract.

Deposit Cash in Alice's Account

We can now deposit cash in Alice's account, using the `CreditAccount` workflow. Alice creates a request to deposit `USD 1000` at the Bank, the Bank then accepts the request and a corresponding *Holding* is created.

```
  aliceRequestCid <- submit alice do
    createCmd CreditAccount.Request with
      account = aliceAccount
      instrument = instrumentKey
      amount = 1000.0

  aliceCashHoldingCid <- submit bank do exerciseCmd aliceRequestCid CreditAccount.
↪Accept
```

You can imagine that the latter step happens only after Alice has shown up at the Bank and delivered physical banknotes corresponding to the amount of the deposit.

The holding contract represents the record of ownership on the ledger. In this scenario, Alice's holding of `1000` units of the cash instrument means that she is entitled to claim `USD 1000` from holding's custodian, the Bank.

To summarize

> an instrument defines *what* a party holds (the rights and obligations).
> a holding defines *how much* (i.e., the amount) of an instrument and *against which party* (i.e., the custodian) the instrument is being held.

Frequently Asked Questions

What are accounts used for?

An account is used as the proof of a business relationship between an owner and a custodian: Alice may transfer cash to Bob because Bob has a valid account at the Bank.

This is done to avoid that Alice transfers cash to Charlie without Charlie being vetted and acknowledged by the Bank.

The account is also used to determine who is required to authorize incoming and outgoing transfers. For the account at hand, the owner acts as a controller for both incoming and outgoing transfers. The other options are explained as part of the settlement tutorials.

Why do we need factories?

You might be wondering why we use account factories and holding factories instead of creating an *Account* or *Holding* directly.

This is done to avoid having to reference the `Daml.Finance.Holding` package directly in the user workflows (and hence simplify upgrading procedures).

This pattern is described in detail in the *Daml Finance Patterns* page and is based on the assumption that there are very few factory contracts which are setup on ledger initialization.

Summary

You now know how to setup basic accounts, holdings, and instruments. The key concepts to take away are:

Holdings represent the ownership of a financial instrument at a custodian.
Instruments define the economic terms of a financial contract.
Accounts ensure that only known parties can obtain ownership.
Factories are used to create the respective contracts without having to depend on implementation packages.

1.21.1.5 Transfer

This tutorial builds on the previous chapter, which introduced *account*, *instrument*, and *holding*.

We are now going to transfer the holding that we created in the previous tutorial from Alice to Bob.

Run the Script

Let us now explore the `Transfer` script step-by-step. It builds on the previous *Holdings* tutorial script in the sense that the same accounts and the existing holdings are used.

Transfer Cash from Alice to Bob

The final step of our `Setup` script transfers Alice's holding to Bob using the `Transfer` workflow. In our tutorial example, the receiver of the cash makes the transfer request:

```
  transferRequestCid <- submit bob do
    createCmd Transfer.Request with
      receiverAccount = bobAccount
      instrument = cashInstrument
      amount = 1000.0
      currentOwner = alice

  bobCashHoldingCid <- submit alice do
    exerciseCmd transferRequestCid Transfer.Accept with holdingCid =
↪aliceCashHoldingCid
```

Bob requests the cash to be transferred to his account. Alice then accepts the request.

You notice that here we make explicit use of the fact that Alice can `readAs` the public party. This is needed as, in order to complete the transfer, visibility on the receiving account's holding factory is required.

Frequently Asked Questions

How does the `Transfer` workflow work?

If you look at the implementation of the `Transfer` workflow, you will notice the following lines:

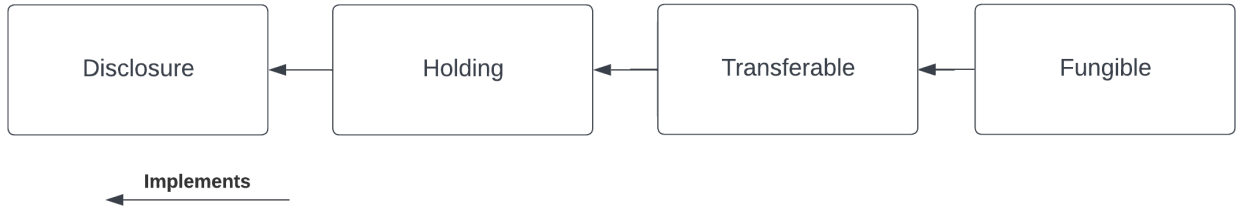
```
        let transferableCid = coerceInterfaceContractId @Transferable.I holdingCid

        newTransferableCid <- exercise transferableCid Transferable.Transfer with
          actors = fromList [currentOwner, receiverAccount.owner]
          newOwnerAccount = receiverAccount

        pure $ toInterfaceContractId @Holding.I newTransferableCid
```

The first line converts the holding contract id (of type *ContractId Holding.I*) to the *Transferable.I* interface using `coerceInterfaceContractId`.

Then, the `Transfer` choice, defined as part of the *Transferable* interface, is invoked.

Finally, the new holding is converted back to a *Holding.I* before it is returned. This is done using `toInterfaceContractId`.

In order to fully understand these instructions, we need to keep in mind the interface hierarchy used by our holding implementation.



We use `coerceInterfaceContractId` to convert the *Holding.I* to a *Transferable*. The success of this operation is not guaranteed and will result in a run-time error if the holding implementation at hand does not implement *Transferable*.

We use `toInterfaceContractId` to convert back to a *Holding*. This is because all *Transferable*s implement the *Holding.I* interface, so the validity of this operation is guaranteed at compile-time.

Why is Alice an observer on Bob's account?

You might have noticed that Alice is an observer of Bob's account and you might be wondering why this is the case.

This is because the party exercising the `Transfer` choice, which in this case is Alice, needs to fetch Bob's account in order to verify that it has not been archived.

If we wanted to avoid Bob's account contract ever being disclosed to Alice, we would need a third party (in this case the Bank) to execute the `Transfer`.

Exercises

There are a couple of improvements to the code that can be implemented as an exercise. They will help you familiarize yourself with the library and with Daml interfaces.

Split the Holding to Transfer the Right Amount

In the example, Bob requests `USD 1000` from Alice and Alice allocates a holding for exactly the right amount, because the transfer would otherwise fail. We want the transfer to be successful also if Alice allocates a holding for a larger amount e.g., `USD 1500`.

We can leverage the fact that the holding implements the *Fungible* interface, which makes it possible to `Split` it into a holding of `USD 1000` and one of `USD 500`. In the implementation of the `CashTransferRequest_Accept` choice:

cast the allocated holding to the *Fungible* interface
use the `Split` choice to split the larger holding into two holdings
execute the transfer, allocating the holding with the correct amount

In the last step, you will need to cast the *Fungible* to a *Transferable* using `toInterfaceContractId`.

Temporary Account Disclosure

There is no reason for Alice to be an observer on Bob's account before the transfer is initiated by Bob (and after the transfer is executed).

Modify the original code, such that:

> Bob's account is disclosed to Alice once the transfer is initiated
> When the Transfer is executed, Alice removes herself from the account observers

In order to do that, you can leverage the fact that *Account* implements the *Disclosure* interface. This interface exposes the `AddObservers` and `RemoveObservers` choices, which can be used to disclose / undisclose Bob's account contract to Alice. In order to exercise these choices, you can use the *Account.exerciseInterfaceByKey* utility function.

Summary

You now learned how to perform a simple transfer. The key concepts to take away are:

> Holdings represent the ownership of a financial instrument at a custodian.
> Transfers change ownership of a holding.

Ownership transfers typically happen as part of a larger financial transaction. The next tutorial will show you how to create such a transaction and how to settle it atomically.

1.21.1.6 Settlement

This tutorial introduces the settlement features of the library through a simple example. The purpose is to demonstrate how multiple holding transfers can be executed atomically.

We are going to:

1. create a new `TOKEN` *instrument*
2. credit a `TOKEN` holding to Alice's account
3. setup a delivery-vs-payment (DvP) transaction to give Alice's `TOKEN` holding to Bob in exchange for a `USD` holding
4. settle this transaction atomically

This example builds on the previous *Transfer* tutorial script in the sense that the same accounts and the existing holdings are used.

Overview of the Process

We first give a quick outline of the settlement process:

1. Define steps to be settled	Two (or more) parties need to first agree on a set of *steps* to be settled.
2. Generate settlement instructions	*Instructions* are generated for each step. An instruction is a contract where the sender can specify its *Allocation* preference for the instruction (e.g., the matching holding they wish to send). The receiver can specify its *Approval* preference for the instruction (e.g., the account they wish to receive the holding to). The creation of Instructions is done by first using a *Route Provider* and then applying a *Settlement Factory*.
3. Allocate and approve instructions	For every instruction, the sender and the receiver specify their allocation and approval preferences, respectively.
4. Settle the batch	A *Batch* contract is used to settle all instructions atomically according to the specified preferences (e.g. by transferring all allocated holdings to the corresponding receiving accounts). This batch contract is created in step 2, together with the settlement instructions.

Run the Script

The code for this tutorial can be executed via the `runSettlement` function in the `Settlement.daml` module.

The first part executes the script from the previous *Transfer* tutorial to arrive at the initial state for this scenario. We then create an additional `TOKEN` *instrument* and credit Alice's account with it.

The interesting part begins once Alice proposes the DvP trade to Bob. Before creating the DvP proposal, we need to instantiate two contracts:

1. *Route Provider*

```
  routeProviderCid <- toInterfaceContractId @RouteProvider.I <$> submit bank
↪do
    createCmd SingleCustodian with
      provider = bank; observers = S.fromList [alice, bob]; custodian = bank
```

   This is used to discover a settlement route, i.e., *routed steps*, for each settlement *step*. In this example, the route provider simply converts each step to a routed step using a single custodian (the bank).
2. *Settlement Factory*

```
  settlementFactoryCid <- toInterfaceContractId @Settlement.F <$> submit bank
↪do
    createCmd Settlement.Factory with
      provider = bank
      observers = S.fromList [alice, bob]
```

   This is used to generate the settlement batch and instructions from the *routed steps*.

Bob creates a `Dvp.Proposal` template to propose the exchange of the `TOKEN` against `USD`.

```
  dvpProposalCid <- submit bob do
    createCmd DvP.Proposal with
      id = "xccy trade"
      recQuantity = qty 10.0 tokenInstrument
      payQuantity = qty 1000.0 usdInstrument
      proposer = bob
      counterparty = alice
      routeProviderCid -- This is equivalent to writing routeProviderCid =
↪routeProviderCid
      settlementFactoryCid
```

Alice then accepts the proposal, agreeing to the terms of the trade.

```
  (batchCid, recSettleInstructionCid, paySettleInstructionCid) <- submit alice do
    exerciseCmd dvpProposalCid DvP.Accept
```

Once the proposal is accepted, three contracts are created:

an instruction to transfer `10 TOKEN` from Alice to Bob
an instruction to transfer `USD 1000` from Bob to Alice
a batch contract to settle the two instructions atomically

The workflow to create these contracts makes use of the route provider and the settlement factory.

```
        (containerCid, [recInstructionCid, payInstructionCid]) <-
          exercise settlementFactoryCid Settlement.Instruct with
            instructors = fromList [proposer, counterparty]
            settlers = singleton proposer
            id = Id id
            description = "Settlement for " <> id
            contextId = None
            routedSteps
            settlementTime = None -- i.e., immediate settlement
```

As a next step, Alice allocates her `TOKEN` holding to the corresponding instruction. Bob then approves the instruction specifying the receiving account.

```
  (allocatedRecSettleInstructionCid, _) <- submit alice do
    exerciseCmd recSettleInstructionCid Instruction.Allocate with
      actors = S.singleton alice
      allocation = Pledge aliceHoldingCid

  approvedRecSettleInstructionCid <- submit bob do
    exerciseCmd allocatedRecSettleInstructionCid Instruction.Approve with
      actors = S.singleton bob
      approval = TakeDelivery bobAccount
```

The same happens in the second instruction (where Bob allocates his `USD` holding and Alice provides the receiving account).

Now that all instructions are fully allocated and approved, they can finally be settled.

```
  [bobHoldingCid, aliceHoldingCid] <- submit bob do
    exerciseCmd batchCid Batch.Settle with
      actors = singleton bob
```

Within the same transaction, Alice receives a `USD` holding from Bob in exchange for a `TOKEN` holding.

Frequently Asked Questions

Why do we need a route provider?

Consider a real-world example where Alice instructs a bank transfer to send USD 100 to Bob. The following happens:

`USD 100` are debited from Alice's account at her bank
`USD 100` are transferred from Alice's bank to Bob's bank (via their accounts at the central bank)
`USD 100` are credited to Bob's account at his bank

A single settlement *Step* requires three *RoutedStep*s to settle.

The same dynamics can be reproduced in Daml with a *Route Provider* implementation, allowing for on-ledger intermediated settlement. For example, see the *Intermediated Lifecycling* tutorial.

Why do we need a settlement factory?

A settlement factory contract is used to generate settlement *Instructions* from *RoutedStep*s. It also generates a *Batch* contract, which is used to settle instructions atomically.

The reason why the factory is needed has already been introduced in the previous tutorial: it provides an interface abstraction, so that your workflow does not need to depend on concrete implementations of *Batch* or *Instructions*.

Can we use a different settler?

In our example, Alice triggers the final settlement of the transaction (by exercising the `Settle` choice on the *Batch* contract).

In principle, a different settler could be chosen. The choice of a settler is usually quite delicate, as this party acquires visibility on the entire transaction and hence needs to be trusted.

What if one party wants to cancel the settlement?

The parties who sign the *Batch* contract (the requestors) can exercise the `Cancel` choice of the *Batch* to cancel all associated *Instructions* atomically.

Summary

You know how to define complex transactions and settle them atomically. The main points to take away are:

A route provider is used to discover settlement routes, i.e., routed steps, for each settlement step.
A settlement factory is used to instruct settlement for an arbitrary list of routed steps.
Instructions are used to collect authorizations, assets to be moved, and means of settlement.
Batches group together instructions to be settled atomically.

In the next tutorial, we will introduce the lifecycling framework of the library, which is used to model the evolution of instruments. The concepts introduced in this tutorial will be used to settle payments arising from lifecycle events.

1.21.1.7 Lifecycling

This tutorial introduces the *lifecycling* framework of the library with a simple example. The purpose is to demonstrate how lifecycle rules and events can be used to process a dividend payment.

We are going to:

1. create a new version of the token instrument
2. create the required lifecycle rules
3. create a distribution event
4. process the event to produce the effects from the distribution
5. instruct settlement by presenting a token holding
6. settling the resulting batch atomically

This example builds on the previous *Settlement* tutorial script in the sense that the same accounts and the existing holdings are used.

Overview of the Process

We first give a high-level outline of the lifecycle process:

1. Create a lifecycle rule	A lifecycle rule implements the logic to calculate effects for a given lifecycle event. In our example we create a distribution rule to handle the dividend event on our token.
2. Create a lifecycle event	The lifecycle event refers to the *target instrument* the event applies to. Holdings on this instrument can then be used to claim the resulting lifecycle effect.
3. Process the event through the lifecycle rule	The lifecycle rule contains the business logic to derive the lifecycle effects resulting from a concrete event. The effect describes the per-unit holding transfers that are to be settled between a custodian and the owner of a holding.
4. Claim the effect using a holding	The claim rule is used to claim the effects resulting from a lifecycle event using a holding on the target instrument. The result is a set of settlement instructions and a corresponding batch to be settled between the custodian and the owner of the holding.

Run the Script

The code for this tutorial can be executed via the `runLifecycling` function in the `Lifecycling.daml` module.

The first part executes the script from the previous *Settlement* tutorial to arrive at the initial state for this scenario.

Then we create a new version of the *token* instrument, which is required for defining the distribution event. This is what the instrument holders will receive when processing the lifecycle event later in the tutorial.

```
  let newTokenInstrument = tokenInstrument with version = "1"
  now <- getTime
  submit bank do
    exerciseCmd tokenFactoryCid Token.Create with
      token = Token with
        instrument = newTokenInstrument
        description = "Instrument representing units of a generic token after the
↪distribution event"
        validAsOf = now
      observers = M.empty
```

Next, we create two lifecycle rules:

```
  distributionRuleCid <- toInterfaceContractId @Lifecycle.I <$> submit bank do
    createCmd Distribution.Rule with
      providers = S.singleton bank
      lifecycler = bank
      observers = S.singleton bob
      id = Id "Lifecycle rule for distribution"
      description = "Rule contract to lifecycle an instrument following a
↪distribution event"
```

(continues on next page)

(continued from previous page)

```
  lifecycleClaimRuleCid <- toInterfaceContractId @Claim.I <$> submitMulti [bank,
↪bob] [] do
    createCmd Claim.Rule with
      providers = S.fromList [bank, bob]
      claimers = S.singleton bob
      settlers = S.singleton bob
      routeProviderCid
      settlementFactoryCid
      netInstructions = False
```

The *Distribution Rule* defines the business logic to calculate the resulting lifecycle effect from a given distribution event. It is signed by the *Bank* as a provider.
The *Claim Rule* allows a holder of the target instrument to claim the effect resulting from the distribution event. By presenting their holding they can instruct the settlement of the holding transfers described in the effect.

We then create a distribution event describing the terms of the dividend to be paid.

```
  distributionEventCid <- toInterfaceContractId @Event.I <$> submit bank do
    createCmd Distribution.Event with
      providers = S.singleton bank
      id = Id "DISTRIBUTION"
      description = "Profit distribution"
      effectiveTime = now
      targetInstrument = tokenInstrument
      newInstrument = newTokenInstrument
      perUnitDistribution = [qty 0.02 usdInstrument]
      observers = S.empty
```

Now we can process the distribution event using the distribution rule.

```
  (_, [effectCid]) <- submit bank do
    exerciseCmd distributionRuleCid Lifecycle.Evolve with
      eventCid = distributionEventCid
      observableCids = []
      instrument = tokenInstrument
```

The result of this is an effect describing the per-unit asset movements to be executed for token holders. Each holder can now present their holding to *claim* the effect and instruct settlement of the associated entitlements.

```
  result <- submit bob do
    exerciseCmd lifecycleClaimRuleCid Claim.ClaimEffect with
      claimer = bob
      holdingCids = [bobHoldingCid]
      effectCid -- This is equivalent to writing effectCid = effectCid
      batchId = Id "DistributionSettlement"
  let [bobInstructionCid, bankInstructionCid, couponInstructionCid] = result.
↪instructionCids
```

As a side-effect of settling the entitlements, the presented holding is exchanged for a holding of the new token version. This is to prevent a holder from benefiting from a given effect twice.

In our example of a cash dividend, only a single instruction is generated: the movement of cash from the bank to the token holder. This instruction along with its batch is settled the usual way, as

described in the previous *Settlement* tutorial.

```
-- Allocate instruction
(bobInstructionCid, _) <- submit bob do
  exerciseCmd bobInstructionCid Instruction.Allocate with
    actors = S.singleton bob
    allocation = Pledge bobHoldingCid

(bankInstructionCid, _) <- submit bank do
  exerciseCmd bankInstructionCid Instruction.Allocate with
    actors = S.singleton bank
    allocation = CreditReceiver

(couponInstructionCid, _) <- submit bank do
  exerciseCmd couponInstructionCid Instruction.Allocate with
    actors = S.singleton bank
    allocation = CreditReceiver

-- Approve instruction
bobInstructionCid <- submit bank do
  exerciseCmd bobInstructionCid Instruction.Approve with
    actors = S.singleton bank
    approval = DebitSender

bankInstructionCid <- submit bob do
  exerciseCmd bankInstructionCid Instruction.Approve with
    actors = S.singleton bob
    approval = TakeDelivery bobAccount

couponInstructionCid <- submit bob do
  exerciseCmd couponInstructionCid Instruction.Approve with
    actors = S.singleton bob
    approval = TakeDelivery bobAccount

-- Settle batch
submit bob do
  exerciseCmd result.batchCid Batch.Settle with actors = S.singleton bob
```

Note that the bank in this case does not actually transfer the cash from another account, but simply credits Bob's account by using the `CreditReceiver` allocation type. In a real-world dividend scenario one would additionally model the flow of funds from the issuer to the bank using the same lifecycle process as described above.

Frequently Asked Questions

Which party should create and sign the lifecycle rules and events?

In the simplified scenario for this tutorial, we have used the bank as both the *issuer* and *depository* for the instruments involved. In a real-world case, instruments and their corresponding lifecycle rules and events would be maintained by an actual issuer, with the depository acting as a 3rd-party trust anchor.

Which parties typically take which actions in the lifecycle workflow?

The lifecycle interfaces governing the process leave the controllers of the various choices in the process up to the implementation.

Typically, we would expect the issuer of an instrument to be responsible to generate lifecycle events (for example, announcing dividends or stock splits).
Lifecycle rules on the other hand are often controlled by 3rd-party calculation agents.
The claiming of lifecycle effects is by default the responsibility of the owner of a holding. If instead the owner wants to delegate this responsibility to their custodian they can do so via a delegation contract.
The party executing settlement can be chosen as well, as described in the previous tutorial on *Settlement*.

Which party should take the role as *lifecycler*?

From a design perspective, a lifecycler is often the party that defines the lifecycle events happening on an instrument (although they can be different). In the simplified example above, it is the bank. In a more realistic example, it would probably be the issuer. In some special cases, if we really need the owner to be the lifecycler, we can use a delegation contract.

The lifecycler is currently trusted with:

Timely and complete Event processing
Providing accurate Observations

Which party is the provider of the Effect?

Most of the time the provider of the Effect is the lifecycler. However, in some cases we may want to avoid disclosing the claimed holdings to the lifecycler. The provider of the Effect gets to see all holdings claimed against that one Effect contract. If we wish to avoid that, we then need a different effect provider.

Can an instrument act as its own lifecycle rule?

Yes, an instrument can implement the `Lifecycle` interface directly such that the lifecycle rules are contained within the instrument itself. There are, however, advantages to separating this logic out into rule contracts:

Keeping lifecycle rules in a different package from your instruments allows you to independently upgrade or patch them without affecting your live instruments.
Having separate rules allows to change the lifecycle properties of an instrument dynamically at runtime. For example, an instrument can initially be created without support for doing asset distributions. Then, at a later point, the issuer might decide to start paying dividends. They can now simply add a distribution rule to the running system to enable this new lifecycle event for their instrument without affecting the actual live instrument itself (or any holdings on it).

Can I integrate a holding ownership change (of the target instrument) within lifecycling?

Lifecycling will not change the ownership of the target instrument. You should use the *Transfer* pattern to do a delivery-versus-payment as a separate step from the lifecycling.

However, there usually is a change of ownership of the other consumed/produced instruments when lifecycling (e.g. when paying out a dividend cash is moved from one party to another).

Summary

You have learned how to use lifecycle rules and events to describe the behavior of an instrument. The key concepts to take away are:

Lifecycle events represent different ways of how an instrument can evolve.
A lifecycle rule contains logic to calculate the effects an event has on an instrument and its holdings.
A claim rule is used to instruct settlement for a given effect using a holding.

1.21.2 Settlement tutorials

This section explains how the settlement processes of Daml Finance work in detail. It combines a step by step description of different workflows with supporting code.

The following tutorials are available:

The *Enhanced Transfers* tutorial builds upon the basic *Transfer* tutorial from the Getting Started section. Specifically, we explore how to configure the controllers that need to authorize incoming transfers (credits) and outgoing transfers (debits) to and from an account, respectively.
The *Internal Settlement* tutorial, an extension of the basic *Settlement* Getting Started tutorial, illustrates how holdings can be transferred within a single custodian through a settlement workflow involving batches and related instructions. The allocation process of such instructions involves methods for committing a pre-existing holding (`Pledge`), a newly created holding (`CreditReceiver`), and a holding received simultaneously (`PassThroughFrom`). The approval methods include taking delivery of a holding to an account (`TakeDelivery`), immediately nullifying the holding (`DebitSender`), and passing the holding through (as allocation) to another instruction (`PassThroughTo`).
The *Intermediated Settlement* tutorial builds upon the *Internal Settlement* tutorial and shows how to make use of a `RouteProvider` to settle instructions across account hierarchies involving more than one custodian.

1.21.2.1 Download the code for the tutorials

As a prerequisite, make sure that the *Daml SDK* is installed on your machine.

Open a terminal and run:

```
daml new finance-settlement --template=finance-settlement
```

This creates a new folder with contents from our template. Navigate to the `finance-settlement` folder and then run the following to download the required Daml Finance packages:

```
./get-dependencies.sh
```

or, if you are using Windows

```
./get-dependencies.bat
```

Finally, you can start Daml Studio to inspect the code and run the project's scripts:

```
daml studio
```

1.21.2.2 Enhanced Transfers

In this tutorial, we delve deeper into the concepts that were introduced in our getting-started tutorials. In particular, we will extend on the *transfer* tutorial.

We begin by understanding the simple form of settlement. It transpires when a customer's funds are transferred to another account within the same bank. Consequently, the sender's account is debited (balance decreases), and the recipient's account is credited (balance increases). This process is internally managed within the bank's systems and usually occurs instantly, as it doesn't require interaction with external systems or institutions.

In Daml Finance, such fund transfers are not necessarily represented by a settlement workflow that involves allocating and approving instructions. Instead, a direct transfer of funds can occur between two parties, such as Alice and Bob. This transfer debits the sending account and atomically credits the receiving account.

Next, we will explore how to configure the controllers responsible for authorizing incoming transfers (credits) and outgoing transfers (debits) of holdings to an account.

Configuring Account Controllers

The *Controllers* data type specifies the parties that need to authorize incoming and outgoing transfers to an account.

For this tutorial, we provide four example scripts illustrating various incoming and outgoing controller settings:

Script	Incoming Controllers	Outgoing Controllers
runDualControlTransfer	Anyone	Both (owner and custodian)
runDiscretionaryTransfer	Custodian	Custodian
runSovereignTransfer	Owner	Owner
runUnilateralTransfer	Anyone	Owner

Each script begins by running a setup script `runSetupTransferRequestWith` that requests a transfer of a holding from Alice to Bob at the Bank. The setup script takes a configuration as input to set up Alice's and Bob's account controllers, as outlined in the table above.

The last step of the setup script creates a transfer request of a holding from Alice to Bob:

```
  let
    transferRequest = Transfer.Request with
      requestor
      receiverAccount = bobAccount
      transferableCid = aliceHoldingCid
      accepted = S.fromList []
      observers = S.fromList [alice, bob, bank]
  transferRequestCid <- submit requestor do createCmd transferRequest
```

The transfer `Request` template is designed for the stepwise collection of the necessary authorizations for transferring a holding to a new owner:

```
template Request
  with
    requestor : Party
      -- ^ The requestor.
    receiverAccount : AccountKey
      -- ^ The account to which the holding is sent.
    transferableCid : ContractId Transferable.I
      -- ^ The holding instance to be sent.
    accepted : Set Party
      -- ^ Current set of parties that accept the transfer.
    observers : Set Party
      -- ^ Observers.
  where
    signatory requestor, accepted
    observer observers

    choice Accept : ContractId Request
      with
        actors : Set Party
      controller actors
      do
        create this with accepted = actors `union` this.accepted

    choice Effectuate : ContractId Transferable.I
      with
        actors : Set Party
      controller actors
      do
        exercise transferableCid Transferable.Transfer with
          actors = actors `union` this.accepted
          newOwnerAccount = receiverAccount
```

Dual Control

In the `runDualControlTransfer` script, both the custodian and the owner of an account must authorize outgoing transfers (debits), while incoming transfers (credits) require no authorization.

This script begins by setting up accounts accordingly and creating a transfer request instance:

```
  let
    dualControl = AccountControllers
        with
```

(continues on next page)

(continued from previous page)

```
        incoming = Anyone
        outgoing = Both
  SetupTransferRequest{bank; alice; bob; requestor; transferRequestCid} <-
    runSetupTransferRequestWith dualControl
```

To execute the transfer, both the Bank and Alice must authorize:

```
  transferRequestCid <- submit bank do
    exerciseCmd transferRequestCid Transfer.Accept with actors = S.singleton bank
  submit alice do
    exerciseCmd transferRequestCid Transfer.Effectuate with actors = S.singleton
↪alice
```

Discretionary

The `runDiscretionaryTransfer` script specifies that the custodian controls both incoming and outgoing transfers:

```
  let
    discretionary = AccountControllers
      with
        incoming = Custodian
        outgoing = Custodian
  setupState@SetupTransferRequest{bank, alice, bob, requestor, transferRequestCid}
↪ <-
    runSetupTransferRequestWith discretionary
```

Following the setup, the Bank can execute the transfer single-handedly:

```
  submit bank do
    exerciseCmd transferRequestCid Transfer.Effectuate with actors = S.singleton
↪bank
```

Sovereign

In the `runSovereignTransfer` script, the owner controls both incoming and outgoing transfers:

```
  let
    sovereign = AccountControllers
      with
        incoming = Owner
        outgoing = Owner
  SetupTransferRequest{bank; alice; bob; requestor; transferRequestCid} <-
    runSetupTransferRequestWith sovereign
```

As Alice is the outgoing controller of the sending account, and Bob is the incoming controller of the receiving account, both need to authorize the transfer:

```
  transferRequestCid <- submit bob do
    exerciseCmd transferRequestCid Transfer.Accept with actors = S.singleton bob
```

(continues on next page)

(continued from previous page)

```
  submit alice do
    exerciseCmd transferRequestCid Transfer.Effectuate with actors = S.singleton
↪alice
```

Unilateral

In our final example script, *runUnilateralTransfer*, the owner controls outgoing transfers, while incoming transfers require no additional authorization:

```
  let
    unilateral = AccountControllers
      with
        incoming = Anyone
        outgoing = Owner
  SetupTransferRequest{bank; alice; bob; requestor; transferRequestCid} <-
    runSetupTransferRequestWith unilateral
```

Once the setup is complete, Alice can independently execute the transfer to Bob:

```
  transferRequestCid <- submit alice do
    exerciseCmd transferRequestCid Transfer.Effectuate with actors = S.singleton
↪alice
```

Summary

By now, you should understand how to configure incoming and outgoing controllers for accounts based on your requirements. Key concepts to remember include:

> To execute a transfer between a sender and a receiver, the outgoing controllers of the sending account and the incoming controllers of the receiving account need to authorize it.
> The required authorization can be provided by a generalized propose-accept template, which allows more than one party to accept.

Ownership transfers usually occur as part of a larger financial transaction. The next tutorials will guide you on how to create such a transaction and how to settle it atomically.

1.21.2.3 Internal Settlement

This tutorial builds upon the concepts introduced in the *Settlement* getting-started tutorial. Compared to our previous *Enhanced Transfers* tutorial, which demonstrated the direct transfer of a holding from sender to receiver (at a single custodian), it delves further into the settlement mechanism with a batch and related instructions. This process enables the adjustment of record books across multiple entities and instrument holdings simultaneously.

In this tutorial, we will limit the complexity by focusing on a single custodian and the transfer of a single instrument. The next tutorial will explore the broader scenario involving multiple custodians. Eager learners are encouraged to extend this tutorial and the following one by incorporating more than one instrument as an exercise.

Understanding Internal Settlement with Examples

To start, let us briefly revisit the settlement process which was explained in the *Settlement* section. A *Batch* consists of one or more *Instructions*. Each instruction signifies a *RoutedStep*, delineating the quantity of an instrument to be transferred from a sender to a receiver at a specific custodian. For an instruction to be prepared for settlement (or execution), the sender-side must furnish an *Allocation*, and the receiver-side must provide an *Approval*.

This tutorial will walk you through three example scripts for settling instructions at a single custodian: `runWrappedTransferSettlement`, `runCreditDebitSettlement`, and `runPassThroughSettlement`.

The allocation processes in these scripts involve methods to commit a pre-existing holding (`Pledge`), a newly created holding (`CreditReceiver`), and a holding received concurrently (`PassThroughFrom`).

The approval methods entail taking delivery of a holding to an account (`TakeDelivery`), immediately nullifying the holding (`DebitSender`), and passing the holding through (as allocation) to another instruction (`PassThroughTo`).

Each script kicks off with `runSetupInternalSettlement` which initiates parties, a cash instrument issued by the Central Bank, accounts for Alice, Bob, and Charlie at a Bank, and a settlement factory:

```
SetupInternalSettlement
  { instrument
  , bank
  , alice, aliceAccount, aliceHoldingCid
  , bob, bobAccount
  , charlie, charlieAccount
  , requestor
  , settlementFactoryCid
  } <- runSetupInternalSettlement
```

The settlement factory is employed by a party, known as the *requestor*, to create a batch and instructions from a list of routed steps. In the scripts, the requestor is also responsible for settling the batch once all instructions have been allocated and approved.

Wrapped Transfer

The first example encapsulates a transfer from Alice to Bob, from our previous *Enhanced Transfers* tutorial, by creating a batch and a single instruction:

```
let
  routedStep = RoutedStep with
    custodian = bank
    sender = alice
    receiver = bob
    quantity = qty 1000.0 instrument

-- Generate settlement instructions from a list of `RoutedStep`s.
(batchCid, [instructionCid]) <-
  submit requestor do
    exerciseCmd settlementFactoryCid Settlement.Instruct with
```

(continues on next page)

(continued from previous page)

```
      instructors = fromList [requestor]
      settlers = singleton requestor
      id = Id "1"
      description = "Transfer from Alice to Bob"
      contextId = None
      routedSteps = [routedStep]
      settlementTime = None -- i.e., immediate settlement
```

Here, Alice allocates by pledging a holding, Bob approves by taking delivery to his account at the Bank, and the requestor finally settles the batch:

```
-- i. Alice allocates.
(instructionCid, _) <- submit alice do
  exerciseCmd instructionCid Instruction.Allocate with
    actors = S.singleton alice
    allocation = Pledge $ toInterfaceContractId aliceHoldingCid

-- ii. Bob approves.
instructionCid <- submit bob do
  exerciseCmd instructionCid Instruction.Approve with
    actors = S.singleton bob
    approval = TakeDelivery bobAccount

-- iii. Requestor executes the settlement.
[bobHoldingCid] <- submit requestor do
  exerciseCmd batchCid Batch.Settle with
    actors = singleton requestor
```

Note that this occurs without involving the Bank, and either Alice or Bob could also take the role as the requestor. As a result of running this script, Alice's holding is transferred to Bob.

Credit and Debit

An alternative approach to transfer the holding from Alice to Bob includes the Bank as an intermediary.

```
let
  routedStep1 = RoutedStep with
    custodian = bank
    sender = alice
    receiver = bank
    quantity = qty 1000.0 instrument
  routedStep2 = routedStep1 with
    sender = bank
    receiver = bob

-- Generate settlement instructions from a list of `RoutedStep`s.
(batchCid, [instructionCid1, instructionCid2]) <-
  submit requestor do
    exerciseCmd settlementFactoryCid Settlement.Instruct with
      instructors = fromList [requestor]
      settlers = fromList [requestor]
      id = Id "1"
```

(continues on next page)

(continued from previous page)

```
      description = "Movement of holding from Alice to Bob through debit and
↪credit"
      contextId = None
      routedSteps = [routedStep1, routedStep2]
      settlementTime = None -- i.e., immediate settlement
```

Similar to the previous scenario, Alice allocates by pledging a holding, and Bob approves by taking delivery to his account. However, in this case, the Bank plays an intermediary role by allocating and approving, debiting the sender and crediting the receiver, respectively:

```
-- i. Alice allocates.
(instructionCid1, _) <- submit alice do
  exerciseCmd instructionCid1 Instruction.Allocate with
    actors = S.singleton alice
    allocation = Pledge $ toInterfaceContractId aliceHoldingCid

-- ii. Bob approves.
instructionCid2 <- submit bob do
  exerciseCmd instructionCid2 Instruction.Approve with
    actors = S.singleton bob
    approval = TakeDelivery bobAccount

-- iii. Bank approves and allocates.
instructionCid1 <- submit bank do
  exerciseCmd instructionCid1 Instruction.Approve with
    actors = S.singleton bank
    approval = DebitSender
(instructionCid2, _) <- submit bank do
  exerciseCmd instructionCid2 Instruction.Allocate with
    actors = S.singleton bank
    allocation = CreditReceiver

-- iv. Requestor executes the settlement.
[bobHoldingCid] <- submit requestor do
  exerciseCmd batchCid Batch.Settle with
    actors = singleton requestor
```

We could have made the Bank approve and allocate its instructions by taking delivery to an account it owns and pledging a holding where it acts as the custodian. However, this would require the creation of dummy accounts and holdings, which can be avoided using the `DebitSender` and `CreditReceiver` methods. These methods can only be used when the receiver (resp. the sender) corresponds to the custodian.

Pass Through

The final script of this tutorial demonstrates how holdings received as part of the same settlement process can be allocated to a subsequent instruction. We again use two instructions, `instruction1` and `instruction2`, but now with Charlie as the intermediary:

```
let
  routedStep1 = RoutedStep with
    custodian = bank
    sender = alice
```

(continues on next page)

(continued from previous page)

```
    receiver = charlie
    quantity = qty 1000.0 instrument
  routedStep2 = routedStep1 with
    sender = charlie
    receiver = bob

-- Generate settlement instructions from a list of `RoutedStep`s.
(batchCid, [instructionCid1, instructionCid2]) <-
  submit requestor do
    exerciseCmd settlementFactoryCid Settlement.Instruct with
      instructors = fromList [requestor]
      settlers = singleton requestor
      id = Id "1"
      description = "Transfer from Alice to Bob via Charlie"
      contextId = None
      routedSteps = [routedStep1, routedStep2]
      settlementTime = None -- i.e., immediate settlement
```

Like in the previous examples, Alice allocates by pledging a holding, and Bob approves by taking delivery to his account. The intermediary, Charlie, allocates using `PassThroughTo (charlieAccount, instruction2)` and approves with `PassThroughFrom (charlieAccount, instruction1)`, essentially enabling the holding Charlie receives from Alice to pass-through to Bob:

```
-- i. Alice allocates.
(instructionCid1, _) <- submit alice do
  exerciseCmd instructionCid1 Instruction.Allocate with
    actors = S.singleton alice
    allocation = Pledge $ toInterfaceContractId aliceHoldingCid

-- ii. Bob approves.
instructionCid2 <- submit bob do
  exerciseCmd instructionCid2 Instruction.Approve with
    actors = S.singleton bob
    approval = TakeDelivery bobAccount

-- iii. Charlie approves and allocates (with pass-through).
instructionKey2 <- retrieveKey charlie instructionCid2
instructionCid1 <- submit charlie do
  exerciseCmd instructionCid1 Instruction.Approve with
    actors = S.singleton charlie
    approval = PassThroughTo (charlieAccount, instructionKey2)
instructionKey1 <- retrieveKey charlie instructionCid1
instructionCid2 <- submit charlie do
  exerciseCmd instructionCid2 Instruction.Allocate with
    actors = S.singleton charlie
    allocation = PassThroughFrom (charlieAccount, instructionKey1)

-- iv. Requestor executes the settlement.
[bobHoldingCid] <- submit requestor do
  exerciseCmd batchCid Batch.Settle with
    actors = singleton requestor
```

The significant advantage of the pass-through method is that Charlie doesn't need any holdings upfront as he's at a net zero position for incoming and outgoing holdings in this settlement process.

Note that the Bank could have utilized the pass-through approach to achieve the same result in the previous script, but it would still require a dummy account.

Summary

By the end of this tutorial, you should have a good grasp on how to apply various allocation and approval methods to instructions. The key points are:

> A custodian can utilize the `DebitSender` and `CreditReceiver` methods to bypass the need for dummy accounts and holdings when approving and allocating instructions, respectively. A holding settled via an intermediary at the same custodian can be passed through, thus eliminating the requirement for the intermediary to possess the holding upfront.

In the forthcoming tutorial, we will delve into more complex settlement transactions involving a transfer across multiple custodians.

1.21.2.4 Intermediated Settlement

This tutorial expands upon the principles discussed in our previous *Internal Settlement* tutorial, providing an in-depth exploration of intermediated settlement involving a multi-level account hierarchy across different custodians.

Understanding Intermediated Settlement with Examples

This tutorial features two example scripts, `runWrappedTransfersSettlement` and `runRouteProviderSettlement`, illustrating how to settle a batch with multiple instructions and custodians.

Each script commences with `runSetupIntermediatedSettlement`, initiating parties, establishing an instrument issued by the Central Bank, and setting up an account hierarchy rooted at the Central Bank. This hierarchy includes two custodian banks, Bank1 and Bank2, and their respective clients Alice, Bob, and Charlie. Holdings are created for Alice@Bank1, Bank1@CentralBank, and Charlie@Bank2:

```
  SetupIntermediatedSettlement
    { instrument
    ; bank1; bank2
    ; alice; aliceAccount; aliceHoldingCid
    ; bob; bobAccount2
    ; charlie; charlieAccount; charlieAccount2; charlieHoldingCid
    ; requestor
    ; settlementFactoryCid
    } <- runSetupIntermediatedSettlement
```

The below diagram illustrates the setup, where edges represent accounts and stars (*) denote holdings:

```
      Central Bank
      */       \
     Bank1     Bank2
    */   \    */   \
Alice   Charlie   Bob
```

Wrapped Transfers: A Detailed Analysis

Our first example elucidates how two transfers at separate custodians can be consolidated into a single batch. The routed steps are as follows:

```
  let
    routedStep1 = RoutedStep with
      custodian = bank1
      sender = alice
      receiver = charlie
      quantity = qty 1000.0 instrument
    routedStep2 = routedStep1 with
      custodian = bank2
      sender = charlie
      receiver = bob
```

These steps are converted into a batch and instructions:

```
  (batchCid, [instructionCid1, instructionCid2]) <-
    submit requestor do
      exerciseCmd settlementFactoryCid Settlement.Instruct with
        instructors = fromList [requestor]
        settlers = singleton requestor
        id = Id "1"
        description = "Transfer from Alice to Bob via Charlie"
        contextId = None
        routedSteps = [routedStep1, routedStep2]
        settlementTime = None -- i.e., immediate settlement
```

The following diagram visualizes this pre- and post-settlement of the batch, where > signifies settlement instructions, and stars (*) represents the holdings:

```
      Central Bank                            Central Bank
      */       \                              */        \
    Bank1      Bank2         "Settle"        Bank1      Bank2
   */   \     */   \            =>          /   \*     /   \*
Alice > Charlie > Bob                    Alice   Charlie   Bob
      T1        T2
```

Similar to the Internal Settlement tutorial, Alice allocates her Bank1 holding through a pledge, while Bob approves its instruction by taking delivery at his account at Bank2. The intermediary, Charlie, approves by taking delivery to his Bank1 account and allocates by pledging his pre-existing holding at Bank2:

```
  -- i. Alice allocates.
  (instructionCid1, _) <- submit alice do
    exerciseCmd instructionCid1 Instruction.Allocate with
      actors = S.singleton alice
      allocation = Pledge $ toInterfaceContractId aliceHoldingCid

  -- ii. Bob approves.
  instructionCid2 <- submit bob do
    exerciseCmd instructionCid2 Instruction.Approve with
      actors = S.singleton bob
      approval = TakeDelivery bobAccount2
```

(continues on next page)

(continued from previous page)

```
  -- iii. Charlie approves and allocates (with pass-through).
  instructionCid1 <- submit charlie do
    exerciseCmd instructionCid1 Instruction.Approve with
      actors = S.singleton charlie
      approval = TakeDelivery charlieAccount
  instructionCid2 <- submit charlie do
    exerciseCmd instructionCid2 Instruction.Allocate with
      actors = S.singleton charlie
      allocation = Pledge $ toInterfaceContractId charlieHoldingCid

  -- iii. Requestor executes the settlement.
  [charlieHoldingCid, bobHoldingCid] <- submit requestor do
    exerciseCmd batchCid Batch.Settle with
      actors = singleton requestor
```

Important to note, Charlie cannot pass-through Alice's holding to Bob, as in the *Internal Settlement* tutorial, due to the holdings having different custodians. Therefore, for settlement to occur, Charlie needs a holding at Bank2. The settlement alters Charlie's counterparty risk, shifting it from Bank1 to Bank2. This is a situation Charlie might wish to avoid. The upcoming example shows how to involve a route provider, eliminating the need for Charlie to hold any upfront holdings, thus preserving his counterparty exposure.

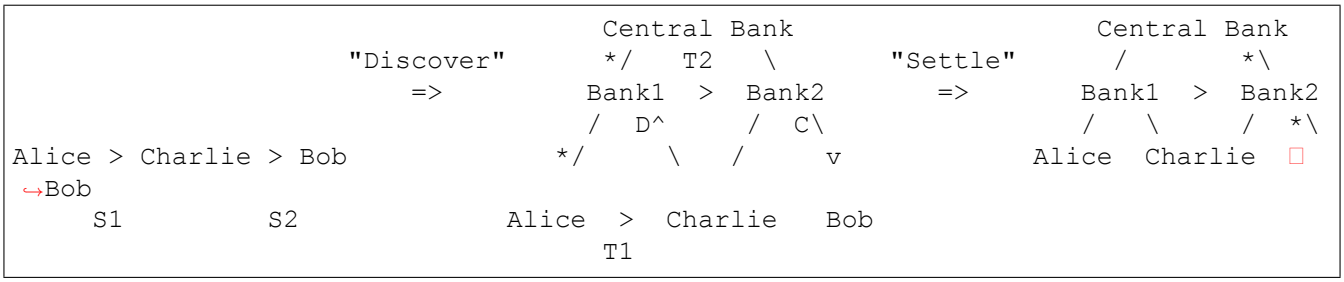
Route Provider: An Alternative Approach

Our second example follows a similar setup to the first, involving a settlement step between Alice and Charlie (S1), and another between Charlie and Bob (S2). However, these steps do not specify a custodian, unlike the routed steps.

To convert the settlement steps S1 and S2 into routed steps, we engage a route provider. The provider suggests preferable routes to the Central Bank for each client. During the Discover action, each step is transformed into a sequence of routed steps

S1 -> (T1)
S2 -> (D, T2, C)

where T1 and T2 denote settlements by transfers, D represents a Debit, and C signifies a Credit. The diagram below depicts the effects of the discovery and settlement process:



Let's begin with the Discover action:

```
  let
    quantity = qty 1000.0 instrument
    -- Alice to Charlie step.
    step1 = Step with sender = alice; receiver = charlie; quantity
```

(continues on next page)

(continued from previous page)

```
    -- Charlie to Bob step.
    step2 = Step with sender = charlie; receiver = bob; quantity

  -- Using a route provider to discover settlement route, with intermediaries,
↪from Alice to Bob for
  -- the instrument.
  let
    paths = M.fromList
      [ ( show instrument.id
        , Hierarchy with
            rootCustodian = cb
            pathsToRootCustodian = [[alice, bank1], [charlie, bank1], [bob,
↪bank2]]
        )
      ]
  routeProviderCid <- toInterfaceContractId @RouteProvider.I <$> submit requestor
↪do
    createCmd IntermediatedStatic with provider = requestor; paths; observers = S.
↪empty
  routedSteps <- submit requestor do
    exerciseCmd routeProviderCid RouteProvider.Discover with
      discoverors = S.singleton requestor; contextId = None; steps = [step1,
↪step2]
  -- Sanity check.
  let
    routedStep1' = RoutedStep with custodian = bank1; sender = alice; receiver =
↪charlie; quantity
    routedStep2' = RoutedStep with custodian = bank1; sender = charlie; receiver
↪= bank1; quantity
    routedStep3' = RoutedStep with custodian = cb; sender = bank1; receiver =
↪bank2; quantity
    routedStep4' = RoutedStep with custodian = bank2; sender = bank2; receiver =
↪bob; quantity
  routedSteps === [routedStep1', routedStep2', routedStep3', routedStep4']
```

Followed by the creation of the batch and instructions:

```
  (batchCid, [instructionCid1, instructionCid2, instructionCid3,
↪instructionCid4]) <-
    submit requestor do
      exerciseCmd settlementFactoryCid Settlement.Instruct with
        instructors = fromList [requestor]
        settlers = singleton requestor
        id = Id "1"
        description = "Transfer from Alice to Bob via intermediaries"
        contextId = None
        routedSteps = routedSteps
        settlementTime = None -- i.e., immediate settlement
```

Finally, Alice, Charlie, Bank1, Bank2, and Bob allocate and approve their instructions accordingly:

```
  -- i. Alice allocates
  (instructionCid1, _) <- submit alice do
    exerciseCmd instructionCid1 Instruction.Allocate with
      actors = S.singleton alice
```

(continues on next page)

(continued from previous page)

```
    allocation = Pledge $ toInterfaceContractId aliceHoldingCid

-- ii. Bob approves.
instructionCid4 <- submit bob do
  exerciseCmd instructionCid4 Instruction.Approve with
    actors = S.singleton bob
    approval = TakeDelivery bobAccount2

-- iii. Charlie approves and allocates.
instruction2 <- retrieveKey charlie instructionCid2
instructionCid1 <- submit charlie do
  exerciseCmd instructionCid1 Instruction.Approve with
    actors = S.singleton charlie
    approval = PassThroughTo (charlieAccount, instruction2)
instruction1 <- retrieveKey charlie instructionCid1
(instructionCid2, _) <- submit charlie do
  exerciseCmd instructionCid2 Instruction.Allocate with
    actors = S.singleton charlie
    allocation = PassThroughFrom (charlieAccount, instruction1)

-- iv. Bank1 approves and allocates.
instructionCid2 <- submit bank1 do
  exerciseCmd instructionCid2 Instruction.Approve with
    actors = S.singleton bank1
    approval = DebitSender
(instructionCid3, _) <- submit bank1 do
  exerciseCmd instructionCid3 Instruction.Allocate with
    actors = S.singleton bank1
    allocation = Pledge $ toInterfaceContractId bank1HoldingCid

-- v. Bank2 approves and allocates.
instructionCid3 <- submit bank2 do
  exerciseCmd instructionCid3 Instruction.Approve with
    actors = S.singleton bank2
    approval = TakeDelivery bank2Account
(instructionCid4, _) <- submit bank2 do
  exerciseCmd instructionCid4 Instruction.Allocate with
    actors = S.singleton bank2
    allocation = CreditReceiver

-- vi. Requestor executes the settlement.
[charlierHoldingCid, bobHoldingCid] <- submit requestor do
  exerciseCmd batchCid Batch.Settle with
    actors = singleton requestor
```

Once the batch is settled, all instructions are executed atomically, causing a coordinated change in the account hierarchy's holdings. Importantly, Charlie acted as an intermediary, providing a route from Alice to Bob, without having to use any holdings upfront.

Summary

You know how to define complex transactions and settle them atomically. Crucial points to remember are:

A route provider serves the purpose of discovering settlement routes or routed steps for each settlement step.
Viewing the account hierarchy as a tree — with the Central Bank at the root, custodians on the second level, and clients as leaves — transfers occur on horizontally directed routed steps, debits on upwards directed routed steps, and credits on downward directed routed steps.

As a challenge for the curious reader, try extending these examples to settle two instruments using settlement routes across two different account hierarchies.

1.21.3 Lifecycling tutorials

This section explains how to lifecycle instruments in Daml Finance. Each tutorial combines a step by step description of different workflows with supporting code.

The following tutorials are available:

The *Time-based lifecycling* tutorial uses a fixed rate bond as an example to demonstrate time-based lifecycling.
The *Observations* tutorial uses a floating rate bond as a sample instrument to show how `Observations` work. This applies to instruments whose payoff depends on an underlying asset.
The *Election-based lifecycling* tutorial uses a callable bond to explain how to create and process elections. This applies to instruments that require an active choice by one of the stakeholders.

Each tutorial builds on top of the previous ones, so they should ideally be followed in order.

1.21.3.1 Download the code for the tutorials

As a prerequisite, make sure that the *Daml SDK* is installed on your machine.

Open a terminal and run:

```
daml new finance-lifecycling --template=finance-lifecycling
```

This creates a new folder with contents from our template. Navigate to the `finance-lifecycling` folder and then run the following to download the required Daml Finance packages:

```
./get-dependencies.sh
```

or, if you are using Windows

```
./get-dependencies.bat
```

Finally, you can start Daml Studio to inspect the code and run the project's scripts:

```
daml studio
```

1.21.3.2 Time-based lifecycling (using a fixed rate bond)

This tutorial describes how to lifecycle instruments with pre-defined payments, e.g. a fixed rate bond. It is similar to the *Lifecycling tutorial*, in that it describes how lifecycle rules and events can be used to evolve instruments over time. However, there is one main difference:

> The *Lifecycling tutorial* describes a dividend event, which is something that the issuer defines *on an ongoing basis*. Only once the date and amount of a dividend payment has been defined, the issuer creates a distribution event accordingly.
> This tutorial describes a fixed rate bond, where all coupon payments are defined *in advance*. They are all encoded in the instrument definition. Hence, the issuer does not need to create distribution events on an ongoing bases. Instead, one lifecycle rule in combination with time events (a date clock) are used to lifecycle the bond.

Check out the *Lifecycling concepts* for a more in-depth description of the evolution of financial instruments over their lifetime.

In this tutorial, we are going to:

1. create a fixed rate bond instrument and book a holding on it
2. create a lifecycle rule
3. create a lifecycle event (time event: `DateClockUpdate`)
4. process the event to produce the effects of a coupon payment
5. instruct settlement by presenting a bond holding
6. settle the resulting batch atomically

This example builds on the previous *Settlement* tutorial script. Some required concepts are explained there, so please check out that tutorial before you continue below.

Run the Script

The code for this tutorial can be executed via the `runFixedRateBond` script in the `FixedRateBond.daml` module.

Instrument and Holding

For the purpose of showcasing time-based lifecycling, we need a suitable sample instrument. *Fixed rate bonds* pay a constant coupon rate at the end of each coupon period. The *Bond Instrument packages* page describes this instrument in more detail. Here, we briefly show how to create the bond instrument using a factory:

```
-- Create a fixed rate bond factory
fixedRateBondFactoryCid <- toInterfaceContractId @FixedRate.F <$> submit bank do
  createCmd FixedRate.Factory with
    provider = bank
    observers = M.empty

-- Define an instrument key for the bond
let
  bondInstrument = InstrumentKey with
    issuer = bank
    depository = bank
    id = Id "FixedRateBond"
```

(continues on next page)

(continued from previous page)

```
    version = "0"
  initialTimestamp = dateToDateClockTime issueDate

-- Bank creates the bond instrument
fixedRateBondCid <- submit bank do
  exerciseCmd fixedRateBondFactoryCid FixedRate.Create with
    fixedRate = FixedRate with
      instrument = bondInstrument
      description = "Instrument representing units of a fixed rate bond"
      couponRate
      periodicSchedule
      holidayCalendarIds
      calendarDataProvider = bank
      dayCountConvention
      currency = usdInstrument
      notional
      lastEventTimestamp = initialTimestamp
    observers = M.fromList pp
```

We also create a bond holding in Bob's account:

```
-- Credit Bob's account with a bond holding
bobRequestCid <- submit bob do
  createCmd CreditAccount.Request with
    account = bobAccount
    instrument = bondInstrument
    amount = 100000.0
bobBondHoldingCid <- submit bank do exerciseCmd bobRequestCid CreditAccount.
↪Accept
```

A holding represents the ownership of a certain amount of an *instrument* by an owner at a custodian. Check out the *Holdings* tutorial for more details.

Now, we have both an instrument definition and a holding. Let us proceed to lifecycle the bond, which is the main purpose of this tutorial.

Lifecycle Events and Rule

As mentioned earlier, we only need one single lifecycle rule to process all time events (since all coupon payments are pre-defined in the instrument terms):

```
-- Create a lifecycle rule
lifecycleRuleCid <- toInterfaceContractId @Lifecycle.I <$> submit bank do
  createCmd Rule with
    providers = S.singleton bank
    observers = M.empty
    lifecycler = bank
    id = Id "LifecycleRule"
    description = "Rule to lifecycle an instrument"
```

In order to lifecycle a coupon payment, we create a time event corresponding to the date of the first coupon:

```
  -- Create a clock update event
  firstCouponClockEventCid <- createClockUpdateEvent bank firstCouponDate S.empty
```

Note that it is the bank that actively creates a *DateClockUpdateEvent*. This results in more control when to actually process the coupon payment. One could also use *LedgerTime*, but that could cause problems in some scenarios, for example:

> The system is down when the coupon should be processed. Processing it the next day is difficult if ledger time is automatically used, because the date returned from the ledger no longer matches the intended lifecycling date.
> A global ledger containing trades from regions in different time zones may require flexibility regarding when, in relation to ledger time, to process coupons and other events.
> The coupon payment depends on market data, which the data provider occasionally provides with a delay. Retroactively processing this is simpler if the lifecycler can provide the *today* date.

Now, we have what we need to actually lifecycle the bond. The `Evolve` choice of the lifecycle rule is exercised to process the time event:

```
  -- Try to lifecycle the instrument
  (lifecycleCid, [effectCid]) <- submit bank do
    exerciseCmd lifecycleRuleCid Lifecycle.Evolve with
      eventCid = firstCouponClockEventCid
      observableCids = []
      instrument = bondInstrument
```

Both the *LedgerTime* and the *DateClock* implement the *TimeObservable* interface, which is used by `Evolve` to specify the current time for the lifecycling.

The result of this is an effect describing the per-unit asset movements to be executed for bond holders. Each holder can now present their holding to *claim* the effect and instruct settlement of the associated entitlements.

A *Claim Rule* allows a holder of the target instrument to claim the effect resulting from the time event:

```
  -- Create the claim rule
  lifecycleClaimRuleCid <- toInterfaceContractId @Claim.I <$> submit bank do
    createCmd Claim.Rule with
      providers = S.fromList [bank]
      claimers = S.singleton bob
      settlers = S.singleton bob
      routeProviderCid
      settlementFactoryCid
      netInstructions = False
```

By presenting their holding they can instruct the settlement of the holding transfers described in the effect:

```
  result <- submitMulti [bob] [public] do
    exerciseCmd lifecycleClaimRuleCid Claim.ClaimEffect with
      claimer = bob
      holdingCids = [bobBondHoldingCid]
      effectCid -- This is equivalent to writing effectCid = effectCid
      batchId = Id "BondSettlement"
  let [bobInstructionCid, bankInstructionCid, couponInstructionCid] = result.
↪instructionCids
```

As a side-effect of settling the entitlements, the presented holding is exchanged for a holding of a new bond version. This is to prevent a holder from benefiting from a given effect twice (in our case: receiving the same coupon twice).

In our example of a bond coupon, only a single instruction is generated: the movement of cash from the bank to the bond holder. This instruction along with its batch is settled the usual way, as described in the previous *Settlement* tutorial.

```
-- Allocate instruction
(bobInstructionCid, _) <- submit bob do
  exerciseCmd bobInstructionCid Instruction.Allocate with
    actors = S.singleton bob
    allocation = Pledge bobBondHoldingCid

(bankInstructionCid, _) <- submit bank do
  exerciseCmd bankInstructionCid Instruction.Allocate with
    actors = S.singleton bank
    allocation = CreditReceiver

(couponInstructionCid, _) <- submit bank do
  exerciseCmd couponInstructionCid Instruction.Allocate with
    actors = S.singleton bank
    allocation = CreditReceiver

-- Approve instruction
bobInstructionCid <- submit bank do
  exerciseCmd bobInstructionCid Instruction.Approve with
    actors = S.singleton bank
    approval = DebitSender

bankInstructionCid <- submit bob do
  exerciseCmd bankInstructionCid Instruction.Approve with
    actors = S.singleton bob
    approval = TakeDelivery bobAccount

couponInstructionCid <- submit bob do
  exerciseCmd couponInstructionCid Instruction.Approve with
    actors = S.singleton bob
    approval = TakeDelivery bobAccount

-- Settle batch
submitMulti [bob] [public] do
  exerciseCmd result.batchCid Batch.Settle with actors = S.singleton bob
```

Note that the bank in this case does not actually transfer the cash from another account, but simply credits Bob's account by using the `CreditReceiver` allocation type. In a real-world bond coupon scenario one would additionally model the flow of funds from the issuer to the bank using the same lifecycle process as described above.

Check out the *Settlement concepts* for a more in-depth description of the different steps in the settlement process and the settlement modes supported by the library.

Note that the lifecycling process above is not limited to fixed coupon bonds. It also works for other instruments with pre-defined payments, for example:

Instrument	Pre-defined variable
Foreign exchange swaps	FX rate
Currency swaps	Interest rates (on both legs)

Frequently Asked Questions

Which party should create and sign the lifecycle rules and events?

In the simplified scenario for this tutorial, we have used the bank as both the *issuer* and *depository* for the instruments involved. In a real-world case, instruments and their corresponding lifecycle rules and events would be maintained by an actual issuer, with the depository acting as a 3rd-party trust anchor.

Which parties typically take which actions in the lifecycle workflow?

The lifecycle interfaces governing the process leave the controllers of the various choices in the process up to the implementation.

Typically, we would expect the issuer of an instrument to define the instrument terms, which in our example above govern the date and amount of each bond coupon.
Lifecycle rules on the other hand are often controlled by 3rd-party calculation agents.
The claiming of lifecycle effects is by default the responsibility of the owner of a holding. If instead the owner wants to delegate this responsibility to their custodian they can do so via a delegation contract.
The party executing settlement can be chosen as well, as described in the previous tutorial on *Settlement*.

Summary

You have learned how to create a fixed coupon bond and how to use a lifecycle rule and events to process the payments pre-defined by the instrument. The key concepts to take away are:

Lifecycle events cause the bond instrument to evolve over time.
A lifecycle rule contains logic to calculate the effects an event has on an instrument and its holdings.
A claim rule is used to instruct settlement for a given effect using a holding.

1.21.3.3 Observations (using a floating rate bond)

This tutorial describes how to define observations. It builds on the previous *Time-based Lifecycling* tutorial, which uses a fixed rate bond where all coupons are pre-defined using a constant annualized rate. In contrast, the coupons of a floating rate bond depend on the value of a reference rate for each coupon period. Hence, the lifecycling framework requires the future values of the reference rate. This is referred to as `Observations`, which is the main topic of this tutorial.

In this tutorial, we are going to:

1. create a floating rate bond instrument and book a holding on it

2. create an observation of the floating rate, which is used to define the coupon payment
3. reuse the lifecycle rule and lifecycle event from the fixed rate bond tutorial
4. process the event to produce the effects of a coupon payment
5. instruct settlement by presenting a bond holding
6. settle the resulting batch atomically

Run the Script

The code for this tutorial can be executed via the `runFloatingRateBond` script in the `FloatingRateBond.daml` module.

Instrument and Holding

For the purpose of showcasing the *Observation* concept, we need a suitable sample instrument. *Floating rate bonds* pay a coupon which is determined by a reference rate, e.g. 3M Libor. The *Bond Instrument packages* page describes this instrument in more detail. Here, we briefly show how to create the bond instrument using a factory:

```
  -- Create a floating rate bond factory
  floatingRateBondFactoryCid <- toInterfaceContractId @FloatingRate.F <$> submit
↪bank do
    createCmd FloatingRate.Factory with
      provider = bank
      observers = M.empty

  -- Define an instrument key for the bond
  let
    bondInstrument = InstrumentKey with
      issuer = bank
      depository = bank
      id = Id "FloatingRateBond"
      version = "0"

  -- Bank creates the bond instrument
  floatingRateBondCid <- submit bank do
    exerciseCmd floatingRateBondFactoryCid FloatingRate.Create with
      floatingRate = FloatingRate with
        instrument = bondInstrument
        description = "Instrument representing units of a floating rate bond"
        referenceRateId
        couponSpread
        periodicSchedule
        holidayCalendarIds
        calendarDataProvider = bank
        dayCountConvention
        currency = usdInstrument
        notional
        lastEventTimestamp = initialTimestamp
      observers = M.fromList pp
```

Compared to the fixed rate bond, notice that this floating rate instrument also has a `referenceRateId`, that specifies which `Observations` to use in the lifecycling section below.

We also create a bond holding in Bob's account:

```
  -- Credit Bob's account with a bond holding
  bobRequestCid <- submit bob do
    createCmd CreditAccount.Request with
      account = bobAccount
      instrument = bondInstrument
      amount = 100000.0
  bobBondHoldingCid <- submit bank do exerciseCmd bobRequestCid CreditAccount.
↪Accept
```

Now, we have both an instrument definition and a holding. Let us proceed to lifecycle the bond using `Observations`, which is the main purpose of this tutorial.

Lifecycle Events and Rule

An *Observation* of a reference rate contains two pieces of information: the interest rate level and the date to which it applies. The rate level can be positive or negative. In our example, we have a negative interest rate:

```
  let observations = M.fromList [(dateToDateClockTime $ date 2019 Jan 16, -0.
↪00311)]

  observableCid <- toInterfaceContractId <$> submit bank do
    createCmd Observation with
      provider = bank; id = Id referenceRateId; observations; observers = M.empty
```

In our case, the bank then creates the observation on the ledger. The *Observation* implements the *NumericObservable* interface, which is used during lifecycling.

In order to lifecycle a coupon payment, we need a lifecycle rule that defines how to process all time events. We also need a time event corresponding to the date of the first coupon. Both of these are the same as in the previous tutorial using a fixed rate bond, so we will reuse them from there.

Now, we have what we need to actually lifecycle the bond:

```
  -- Try to lifecycle the instrument
  (lifecycleCid, [effectCid]) <- submit bank do
    exerciseCmd lifecycleRuleCid Lifecycle.Evolve with
      eventCid = firstCouponClockEventCid
      observableCids = [observableCid]
      instrument = bondInstrument
```

The difference compared to the previous tutorial is that here we also pass in the observables to the Evolve choice. They are used to evaluate the reference rate on the relevant fixing date of the coupon payment currently being lifecycled.

The result of this is an effect describing the per-unit asset movements to be executed for bond holders. Each holder can now present their holding to *claim* the effect and instruct settlement of the associated entitlements.

The remaining steps (define a claim rule, claim the effect and settle the entitlements) are identical to the previous tutorial.

Note that the lifecycling process above is not limited to floating rate bonds, which have a reference rate as observable. It also works for other instruments that depend on an underlying asset, for example:

Instrument	Observed variable
Inflation linked bond	Inflation index
Interest rate swap	Reference rate (similar to a floating rate bond)
Asset swap	Reference asset
Credit default swap	Default probability & Recovery rate (two observables)
Vanilla option	Reference asset (often end of day fixing)
Barrier option	Reference asset (often intraday observations)

Frequently Asked Questions

Which party should create the observations?

In the simplified scenario for this tutorial, the bank created the observation. In a real-world case, it would probably be the issuer (or a 3rd-party reference data agent) that creates the observations.

Summary

You have learned how to create a floating rate bond and how to define observations that define the amount of the coupon payments. The key concepts to take away are:

Observations are required in order to lifecycle some instruments.
Observations are a general concept that can be used to model different kind of payoffs, using various types of underlyings.
Lifecycling instruments with observations works in a very similar manner compared to those without.

1.21.3.4 Election-based lifecycling (using a callable bond)

This tutorial describes how to define and process Elections. It builds on the previous *Time-based Lifecycling* tutorial, which uses a fixed rate bond where all coupons are pre-defined and are paid out as time passes. In contrast, the coupons of a callable bond depend on whether the issuer has called the bond. Hence, a simple time event is not sufficient to define the next state of the instrument. Instead, the lifecycling framework requires an active Election to be made on each call date. This Election is the main topic of the tutorial. Check out the *Lifecycling concepts* for more details on time based vs election based evolution of instruments.

In this tutorial, we are going to:

1. create a callable bond instrument and book a holding on it
2. reuse the lifecycle rule and settlement factory from the fixed rate bond tutorial
3. create the election not to call the bond
4. process the election event to produce the effects of a coupon payment
5. instruct settlement by presenting a bond holding
6. settle the resulting batch atomically

Run the Script

The code for this tutorial can be executed via the `runCallableBond` script in the `CallableBond.daml` module.

Instrument and Holding

In order to demonstrate the *Election* concept, we need a suitable sample instrument. *Callable bonds* pay coupons as long as the bond has not been called by the issuer. The *Bond Instrument packages* page describes this instrument in more detail. Here, we briefly show how to create the bond instrument using a factory:

```
-- Create a callable bond factory
callableBondFactoryCid <- toInterfaceContractId @Callable.F <$> submit bank do
  createCmd Callable.Factory with
    provider = bank
    observers = M.empty

-- Define an instrument key for the bond
let
  bondInstrument = InstrumentKey with
    issuer = bank
    depository = bank
    id = Id "CallableBond"
    version = "0"

-- Bank creates the bond instrument
callableBondCid <- submit bank do
  exerciseCmd callableBondFactoryCid Callable.Create with
    callable = Callable with
      instrument = bondInstrument
      description = "Instrument representing units of a callable bond"
      floatingRate
      couponRate
      capRate
      floorRate
      couponSchedule = periodicSchedule
      noticeDays
      callSchedule = periodicSchedule
      holidayCalendarIds
      calendarDataProvider = bank
      dayCountConvention
      useAdjustedDatesForDcf
      currency = usdInstrument
      notional
      lastEventTimestamp = initialTimestamp
      prevElections = []
    observers = M.fromList pp
```

Compared to the fixed rate bond, notice that this callable instrument also has a `callSchedule`, that specifies the dates on which the issuer can call the bond.

We also create a bond holding in Bob's account:

```
  -- Credit Bob's account with a bond holding
  bobRequestCid <- submit bob do
    createCmd CreditAccount.Request with
      account = bobAccount
      instrument = bondInstrument
      amount = 100000.0
  bobBondHoldingCid <- submit bank do exerciseCmd bobRequestCid CreditAccount.
↪Accept
```

Now, we have both an instrument definition and a holding. Let us proceed to lifecycle the bond using `Elections`, which is the main purpose of this tutorial.

Lifecycle Events and Rule

We start by creating an Election factory, which can be used to create elections:

```
  -- Create election factory to allow holders to create elections
  electionFactoryCid <- submit bank do
    toInterfaceContractId @Election.F <$> createCmd Election.Factory with
      provider = bank
      observers = M.fromList [("Observers", S.fromList [bob, bank])]
```

An *Election* contains three main pieces of information:

the election tag (e.g. CALLED)
who is making the election (e.g. the bank)
the date to which it applies.

In our example, the bank chooses *not* to call the bond:

```
  -- Create an Election for the first coupon date: do not call the bond.
  electionCid <- submit bank do
    exerciseCmd electionFactoryCid Election.Create with
      actors = S.singleton bank
      id = Id "election id"
      description = "election for a callable bond"
      claim = "NOT CALLED"
      electionTime = dateToDateClockTime $ date 2019 May 15
      electorIsOwner = False
      elector = bank
      counterparty = bank
      instrument = bondInstrument
      amount = 100000.0
      observers = M.fromList [("Holders", S.fromList [bank, bob])]
      provider = bank
```

Note the flag *electorIsOwner* above. Since the bank is not the owner of the bond holding, this flag is *False* in our example. On the other hand, if an investor Alice would have had a holding in a puttable bond, the election whether or not to put would have belonged to Alice (the holding owner), so this flag would have been *True*.

Also, note that there is an *amount* in the election above. This allows the elector to create an election for a specific number of holding units.

Now, we have what we need to actually lifecycle the bond. The `Apply` choice is exercised in order to process the election:

```
  -- Apply election to generate new instrument version + effects
  (newInstrumentKey, [effectCid]) <- submit bank do
    exerciseCmd electionCid Election.Apply with
      observableCids = []
      exercisableCid = coerceInterfaceContractId @Election.Exercisable
↪lifecycleRuleCid
```

In order to lifecycle the coupon payment above, we need a lifecycle rule that defines how to process all election events. The lifecycle rule from the previous tutorial can be reused for this, if we first convert it to an `Election.Exercisable`, as described above.

A *Claim Rule* allows the elector to claim the effect resulting from the election event:

```
  -- Create a new claim rule with the bank as claimer, since it is the bank that
↪does the election.
  lifecycleClaimRuleCid <- toInterfaceContractId @Claim.I <$> submit bank do
    createCmd Claim.Rule with
      providers = S.fromList [bank]
      claimers = S.singleton bank
      settlers = S.singleton bob
      routeProviderCid
      settlementFactoryCid
      netInstructions = False
```

Note that even though we already had a claim rule in the previous example, we could not reuse it because that one was for the *holding owner* to claim the results, whereas in the case of Election based lifecycling it is the *elector* that should claim them:

```
  -- Claim effect
  result <- submit bank do
    exerciseCmd lifecycleClaimRuleCid Claim.ClaimEffect with
      claimer = bank
      holdingCids = [bobBondHoldingCid]
      effectCid
      batchId = Id "BondSettlement"
  let [bobInstructionCid, bankInstructionCid, couponInstructionCid] = result.
↪instructionCids
```

The result of this is an effect describing the per-unit asset movements to be executed for bond holders.

The remaining steps (settling the entitlements) are identical to the previous tutorial.

Note that the election process above is not limited to callable bonds. It also works for other instruments that require a manual decision, such as a physically settled option with a manual exercise decision.

Frequently Asked Questions

Which party should create the elections?

This depends on the economics of the instrument. For example, in a callable bond, it is the issuer of the bond that has the right to choose whether or not to call the bond on the call dates. On the other hand, in the case of a puttable bond, it would be the investor that can elect to demand early repayment of the bond.

What if a bond can only be called on some coupon dates?

Some instruments can require both time based and election based lifecycling. For example, consider a callable bond that has a quarterly coupon but a call schedule that only allows the bond to be called once a year. In this case, an Election has to be created on the call dates to lifecycle the bond. On the other coupon dates, regular time based lifecycling is required to process the coupon payments.

Summary

You have learned how to create a callable bond and how to define Elections to choose whether or not to call the bond. The key concepts to take away are:

Elections are required in order to lifecycle some instruments that require an *active choice* by one of the stakeholders.
Depending on the economics of the instrument, either the holding owner or the issuer should create the election.
Some instruments require both time based and election based lifecycling.

1.21.4 Payoff Modeling tutorials

This section contains an introduction to the Daml Finance *Generic Instrument*, which provides a flexible framework to structure custom payoffs and lifecycle them on the ledger.

The Generic Instrument encapsulates the *Contingent Claims* library, which models the economic terms of an instrument based on its future cashflows.

The tutorials introduce the Contingent Claims modeling framework in a practical way and give you the tools to

structure financial instruments such as bonds, swaps, options, and other derivatives
lifecycle the instruments on-ledger to calculate pending payments

The following tutorials are available:

The *Basic Builders* tutorial introduces the basic Contingent Claims builders.
The *Observations* tutorial shows how to model market observables, such as interest rates or equity spot prices.

1.21.4.1 Download the code for the tutorials

As a prerequisite, make sure that the *Daml SDK* is installed on your machine.

Open a terminal and run:

```
daml new finance-payoff-modeling --template=finance-payoff-modeling
```

This creates a new folder with contents from our template. Navigate to the `finance-payoff-modeling` folder and then run the following to download the required Daml Finance packages:

```
./get-dependencies.sh
```

or, if you are using Windows

```
./get-dependencies.bat
```

Finally, you can start Daml Studio to inspect the code and run the project's scripts:

```
daml studio
```

1.21.4.2 Basic builders

This tutorial introduces the basic claim constructors and shows how to use them to describe a payoff in terms of the future cashflows between the claim's owner and their counterparty. At the end of this section, you should be able to model payoffs such as fixed rate bonds and FX forwards.

You can use the `PayoffBuilder` module to follow along and test the claims described below. When you run the `runCreateAndLifecycle` script, it will

create a Daml Finance Generic instrument wrapping your input claim
lifecycle the instrument at the specified dates
print out pending cashflows

Builders

Zero

The `zero` constructor is used to indicate the absence of cashflows. We can setup this very simple initial payoff as follows

```
    c = zero

    acquisitionDate = date 2023 Aug 01
```

The acquisition date is used to track the date at which two parties enter the contract and it is a required input to each claim.

One

The `one` constructor is used to deliver to the owner of the contract one unit of a specified instrument. For instance, the claim

```
    c = one "USD"

    acquisitionDate = date 2023 Aug 01
```

gives the owner an immediate right to receive one unit of the `USD` instrument.

We can verify that by lifecycling the claim: we define a set of lifecycle dates

```
    lifecycleDates =
      [
        date 2023 Aug 01
      , date 2023 Aug 03
      ]
```

and run the script to obtain

```
"--- EFFECT on 2023-08-01 ---"
"TARGET INSTRUMENT : MyClaim version 0"
"RECEIVING"
" => 1.0 USD"
"GIVING"
```

When we lifecycle as of `01 Aug 2023` a payment of 1 `USD` is received by the owner. This is recorded in the corresponding `Effect` contract. The claim then becomes worthless (it becomes the `zero` claim) and any subsequent lifecycling yields no additional effects.

Scale

The `scale` constructor is used to multiply a claim's cashflows by a certain factor.

```
    c = scale (Const 100.0) $ one "USD"

    acquisitionDate = date 2023 Aug 01
```

As expected, lifecycling now yields

```
"--- EFFECT on 2023-08-01 ---"
"TARGET INSTRUMENT : MyClaim version 0"
"RECEIVING"
" => 100.0 USD"
"GIVING"
```

Give, And

The `and` constructor is used to sum cashflows from multiple sub-claims.

`give` is used to exchange rights and obligations, flipping the direction of cashflows.

We can define a very simple FX trade as follows, where the owner receives `EUR` in exchange for `USD`.

```
    c1 = scale (Const 90.0) $ one "EUR"
    c2 = scale (Const 100.0) $ one "USD"
    c = c1 `and` (give c2)

    acquisitionDate = date 2023 Aug 01
```

When the claim is lifecycled, we obtain

```
"--- EFFECT on 2023-08-01 ---"
"TARGET INSTRUMENT : MyClaim version 0"
"RECEIVING"
" => 90.0 EUR"
"GIVING"
" => 100.0 USD"
```

When you want to additively combine more than two claims, you can use the `andList` constructor.

When

The `when` constructor is used to introduce a time shift, delaying the acquisition of another claim to a point in the future when a certain predicate is met. For instance, the claim

```
    maturity = date 2023 Aug 31
    c = when (at maturity) $ one "USD"

    acquisitionDate = date 2023 Aug 01
```

pays one `USD` once the maturity date is reached, but not before.

When this is lifecycled before maturity, no effect is generated. On the other hand, once we reach maturity we observe

```
"--- EFFECT on 2023-08-31 ---"
"TARGET INSTRUMENT : MyClaim version 0"
"RECEIVING"
" => 1.0 USD"
"GIVING"
```

The `at` function is used to construct a predicate which becomes `True` exactly at the input date, triggering the acquisition of the sub-claim `one "USD"`.

Structuring financial instruments

Equipped with these basic claim builders, we can already structure a variety of real-world financial instruments.

Fixed Rate Bond

A fixed rate bond pays a fixed interest rate over its term and repays the principal amount at maturity. This can be represented as follows

```
interestAmount = Const 50.0
principal = Const 100000.0
c = andList [
    when (at d1) $ scale interestAmount $ one "USD"
  , when (at d2) $ scale interestAmount $ one "USD"
  , when (at maturity) $ scale (interestAmount + principal) $ one "USD"
  ]
```

for `d1 ≤ d2 ≤ maturity`.

FX Forward

As an exercise, try to model an FX Forward, which is a contractual agreement between two parties to exchange a pair of currencies at a set rate on a future date.

Summary

You have learned the basic claim constructors and are now able to structure some real-world financial instruments. The next tutorial will introduce `Observations`, which are used to model time-dependent market observables, such as stock prices and interest rates fixings.

1.21.4.3 Observations

Equity forward payoff

The payoff of financial derivatives depends on the performance of a certain underlying in the future. As an example, consider an equity forward contract where a party agrees to purchase one stock of Apple at a predetermined date and price in the future. The contract buyer will make a profit if the market price of the stock is greater than the purchase price they pay (and make a loss otherwise).

This payoff can be written as follows

```
    maturity = date 2023 Aug 31
    c = when (at maturity) $ scale (Observe "AAPL" - Const 195.0) $ one "USD"

    acquisitionDate = date 2023 Aug 01
```

In this example, the predetermined purchase price (strike price) is `195 USD`.

`Observe "AAPL"` is used to represent the time-dependent market price of Apple, which is unknown at trade inception.

When lifecycling the claim at maturity, we must provide the observed market value for the observable `AAPL` in order to resolve the claim's cashflows.

```
    observations =
      [
        ("AAPL", [(date 2023 Aug 31, 200.0)])
      ]
```

This yields the expected payoff

```
"--- EFFECT on 2023-08-31 ---"
"TARGET INSTRUMENT : MyClaim version 0"
"RECEIVING"
" => 5.0 USD"
"GIVING"
```

You can change the stock's observed value in the script and see how this impacts the generated cashflows.

As you might have realised, the multiplying factor within a `scale` builder does not have to be a constant or deterministic quantity. It is a generic `Observation`, which is a combination of known amounts (built with `Const`) and market observables (built with `Observe`). These building blocks can be combined together using standard algebraic operations (+, −, * , /).

In the example above, once the maturity date is reached the sub-claim `scale (Observe "AAPL" - Const 195.0) $ one "USD"` is acquired and the value of the observation is looked up in the table for that maturity date.

Floating Rate Note

There are cases where we want to explicitly specify the date at which a market observable is evaluated. Take for example the case of a Floating Rate Note, which is a financial instrument that pays a floating coupon based on an interest rate value observed a few days earlier.

An example of such payoff is the following

```
    maturity = date 2023 Aug 31
    observation =  ObserveAt "USD_LIBOR_3M" (date 2023 Aug 10) * Const 1000000.0
    c = when (at maturity) $ scale observation $ one "USD"

    acquisitionDate = date 2023 Aug 01
```

where

> we observe the value of the 3 month US Dollar LIBOR rate on `10 Aug 2023`
> we pay a coupon for that rate on `31 Aug 2023`

The `ObserveAt` observation builder is used to specify when the rate should be observed. In order to lifecycle the claim at maturity we must include the rate observation in the table

```
    observations =
      [
```

(continues on next page)

(continued from previous page)

```
        ("USD_LIBOR_3M", [(date 2023 Aug 10, 0.0563)])
      ]
```

which then yields to the expected payout

```
"--- EFFECT on 2023-08-31 ---"
"TARGET INSTRUMENT : MyClaim version 0"
"RECEIVING"
" => 56300.0 USD"
"GIVING"
```

Interest Rate Swap

As an exercise, try to model

a fixed-for-floating interest rate swap, where the claim owner receives coupons based on a floating rate in exchange for fixed rate coupons.
a basis rate swap, where the owner receives coupons based on 3 month US Dollar LIBOR and pays coupons based on 6 month US Dollar LIBOR to their counterparty

Summary

You now know how to setup payoffs containing complex time-dependent market observables. You have the tools to model a large set of financial products, such as forwards and most interest rate swap variations.

1.21.5 Advanced Topics

This section covers some advanced Daml Finance topics, helping you model complex use-cases and scenarios.

The following tutorials are available:

The *Intermediated Lifecycling* tutorial demonstrates how to lifecycle an instrument with an intermediary party between the issuer and the investor.
The *Contingent Claims Instrument* tutorial describes how to leverage the Contingent Claims library to lifecycle custom instrument implementations.

1.21.5.1 Intermediated Lifecycling of an Instrument

This tutorial describes the *lifecycling* flow of an instrument with an intermediary party between the issuer and the investor. We will use the a *Generic Instrument*, but the same concepts apply to other instrument types as well.

We will illustrate the following steps:

1. Creating a Generic Instrument modeling a fixed rate bond
2. Defining an intermediated settlement route
3. Defining a suitable lifecycle event
4. Lifecycling the instrument

5. Non-atomic settlement of the lifecycle effects
6. Atomic settlement of the lifecycle effects

To follow the script used in this tutorial, you can clone the Daml Finance repository. In particular, the file Instrument/Generic/Test/Intermediated/BondCoupon.daml is the starting point of this tutorial. It contains an example for both non-atomic and atomic settlement of lifecycle effects. In this tutorial we will focus on the non-atomic settlement, but we will mention atomic settlement towards the end.

Create the Instrument

We start by using a *Generic Instrument* to model a fixed rate bond paying a 4% p.a. coupon with a 6M coupon period.

Define an Intermediated Settlement Route

In the case of intermediated lifecycling, we need to define a settlement route for the bond instrument, which depends on the account structure:

```
    {-
    Bond (security) account structure :

      Issuer
        |
        CSD
        |
      Investor
    -}
    let
      route =
        ( bondLabel
        , Hierarchy with
            rootCustodian = issuer
            pathsToRootCustodian = [[investor, csd]]
        )
```

Similarly, we define a settlement route for the cash instrument instrument:

```
      {-
        Cash account structure :

            Central Bank
            /    |     \
          CSD  Issuer  Bank
                         \
                       Investor
      -}
      route =
        ( label
        , Hierarchy with
            rootCustodian = centralBank
            pathsToRootCustodian = [[investor, bank], [csd], [issuer]]
        )
```

Define a Lifecycle Event

Since the bond pays a coupon twice a year, payment is a time-based event. The requirement to pay the coupon is governed by actual time. However, in a trading and settlement system, it is useful to be able to control the time variable, in order to simulate previous/future payments, or to have some flexibility regarding when to process events.

Because of this, the issuer defines a clock update event contract, which signals that a certain time has been reached:

```
-- create clock update event
clockEventCid <- createClockUpdateEvent (S.singleton issuer) today S.empty
```

Lifecycle the Bond Instrument

Using the *Lifecycle* interface, the CSD creates a lifecycle rule contract:

```
-- Create a lifecycle rule
lifecycleRuleCid <- toInterfaceContractId @Lifecycle.I <$> submit csd do
  createCmd Lifecycle.Rule with
    providers = S.singleton csd
    observers= M.empty
    lifecycler = issuer
    id = Id "LifecycleRule"
    description = "Rule to lifecycle a generic instrument"
```

The issuer of the bond is responsible for initiating the lifecycling of the coupon payment, by exercising the `Evolve` choice on the coupon date:

```
-- Try to lifecycle the instrument
(_, [effectCid]) <- submit issuer do
  exerciseCmd lifecycleRuleCid Lifecycle.Evolve with
    eventCid = clockEventCid
    observableCids = []
    instrument = bondInstrument
```

This internally uses the *Event* interface. In our case, the event is a clock update event, since the coupon payment is triggered by the passage of time.

The return type of `effectCid` is an *Effect* interface. It will contain the effect(s) of the lifecycling, in this case a coupon payment. If there is nothing to lifecycle, for example because there is no coupon to be paid today, this would be empty.

Non-atomic Settlement

In order to process the effect(s) of the lifecycling (in this case: pay the coupon), we need to create settlement instructions. In the non-atomic case, this is done in two steps.

First, there is the settlement between the issuer and the CSD. By using the `EffectSettlementService` template, the issuer can claim and settle the lifecycling effects in one step by exercising the `ClaimAndSettle` choice:

```
  -- Setup settlement contract between issuer and CSD
  -- In order for the workflow to be successful, we need to disclose the CSD's
↪cash account to the
  -- Issuer.
  Account.submitExerciseInterfaceByKeyCmd @Disclosure.I [csd] [] csdCashAccount
    Disclosure.AddObservers with
      disclosers = S.singleton csd; observersToAdd = ("Issuer", S.singleton
↪issuer)

  settle1Cid <- submitMulti [csd, issuer] [] do
    createCmd EffectSettlementService with
      csd
      issuer
      instrumentId = bondInstrument.id
      securitiesAccount = csdAccountAtIssuer
      issuerCashAccount
      csdCashAccount
      settlementRoutes = routes

  -- CSD claims and settles effect against issuer
  (effectCid, newInstrumentHoldingCid, [cashHolding]) <- submitMulti [issuer]
↪[publicParty] do
    exerciseCmd settle1Cid ClaimAndSettle with
      instrumentHoldingCid = csdBondHoldingCid; cashHoldingCid =
↪issuerCashHoldingCid; effectCid
```

Then, there is the settlement between the CSD and the investor. We start by creating a settlement factory:

```
  -- investor claims effect against CSD
  routeProviderCid <- toInterfaceContractId <$> submit csd do
    createCmd IntermediatedStatic with
      provider = csd; observers = S.singleton investor; paths = routes

  settlementFactoryCid <- submit csd do
    toInterfaceContractId <$> createCmd Factory with
      provider = csd; observers = S.singleton investor
```

Settlement instructions are created by using the *Claim* interface and exercising the `ClaimEffect` choice:

```
  lifecycleClaimRuleCid <- toInterfaceContractId @Claim.I <$> submit csd do
    createCmd Claim.Rule with
      providers = S.singleton csd
      claimers = S.fromList [csd, investor]
      settlers
      routeProviderCid
      settlementFactoryCid
      netInstructions = False

  result <- submit csd do
    exerciseCmd lifecycleClaimRuleCid Claim.ClaimEffect with
      claimer = csd
      holdingCids = [investorBondHoldingCid]
      effectCid
      batchId = Id "CouponSettlement"
```

Claiming the effect has two consequences:

the investor's holding is upgraded to a new instrument version (where the coupon has been paid)
settlement instructions are generated in order to process the coupon payment

Finally, the settlement instructions are allocated, approved and then settled.

```
  let
    [investorBondInstructionCid, csdBondInstructionCid, csdCashInstructionCid,
      bankCashInstructionCid] = result.instructionCids

  -- Allocate instructions
  (investorBondInstructionCid, _) <- submit investor do
    exerciseCmd investorBondInstructionCid Instruction.Allocate with
      actors = S.singleton investor; allocation = Pledge investorBondHoldingCid
  (csdBondInstructionCid, _) <- submit csd do
    exerciseCmd csdBondInstructionCid Instruction.Allocate with
      actors = S.singleton csd; allocation = CreditReceiver
  (csdCashInstructionCid, _) <- submit csd do
    exerciseCmd csdCashInstructionCid Instruction.Allocate with
      actors = S.singleton csd; allocation = Pledge cashHolding
  (bankCashInstructionCid, _) <- submit bank do
    exerciseCmd bankCashInstructionCid Instruction.Allocate with
      actors = S.singleton bank; allocation = CreditReceiver

  -- Approve instructions
  investorBondInstructionCid <- submit csd do
    exerciseCmd investorBondInstructionCid Instruction.Approve with
      actors = S.singleton csd; approval = DebitSender
  csdBondInstructionCid <- submit investor do
    exerciseCmd csdBondInstructionCid Instruction.Approve with
      actors = S.singleton investor; approval = TakeDelivery
↪investorSecuritiesAccount
  csdCashInstructionCid <- submit bank do
    exerciseCmd csdCashInstructionCid Instruction.Approve with
      actors = S.singleton bank; approval = TakeDelivery bankCashAccount
  bankCashInstructionCid <- submit investor do
    exerciseCmd bankCashInstructionCid Instruction.Approve with
      actors = S.singleton investor; approval = TakeDelivery investorCashAccount

  -- Settle batch
  [investorCashHoldingCid, bankCashHoldingCid, investorBondHoldingCid] <-
    submitMulti (S.toList settlers) [publicParty] do
      exerciseCmd result.batchCid Batch.Settle with actors = settlers
```

Following settlement, the investor receives a cash holding for the due coupon amount.

Atomic Settlement

In the non-atomic settlement case above, settlement was done in two steps: first from issuer to CSD and then from CSD to investor. In atomic settlement, this is done in on step.

The first part of the process is very similar. The first important difference is when the CSD exercises the `ClaimEffect` choice, where the bond holdings of both the CSD and the investor are provided:

```
result <- submit csd do
  exerciseCmd lifecycleClaimRuleCid Claim.ClaimEffect with
    claimer = csd
    holdingCids = [csdBondHoldingCid, investorBondHoldingCid]
    effectCid
    batchId = Id "CouponSettlement"
```

There are now more settlement instructions (both from CSD to issuer and from issuer to CSD):

```
let
  [   csdBondInstructionCid1       -- old bond from CSD to issuer
    , investorBondInstructionCid   -- old bond from investor to CSD
    , issuerBondInstructionCid     -- new bond from issuer to CSD
    , csdBondInstructionCid2       -- new bond from CSD to investor
    , issuerCashInstructionCid     -- coupon payment from issuer to CSD
    , csdCashInstructionCid        -- coupon payment from CSD to investor's bank
    , bankCashInstructionCid       -- coupon payment from investor's bank to
↪investor
    ] = result.instructionCids
```

These will have to be allocated, approved and settled similarly to the non-atomic case above. See the file Instrument/Generic/Test/Intermediated/BondCoupon.daml for full details.

Frequently Asked Questions

What if one party wants to cancel the settlement?

The parties who sign the *Batch* contract (the requestors) can exercise the `Cancel` choice of the *Batch* to cancel all associated *Instructions* atomically.

Parties who are not a requestor can prevent settlement by not approving / allocating their instructions (since a batch is only successful if the settlement instructions are fully allocated and approved).

1.21.5.2 How to leverage Contingent Claims in custom instrument implementations

They *Payoff Modeling tutorial* introduces the *Contingent Claims* modeling framework in the context of the *Generic Instrument*. In this chapter, we will see how the library can instead be used to lifecycle custom instrument implementations. The *Bond* and *Swap* instruments, for example, leverage Contingent Claims behind the scenes to calculate pending coupon payments.

Let us explore in detail how the *fixed rate bond instrument* is implemented in Daml Finance. The goal is for you to learn how to implement and lifecycle your own instrument template, should you need an instrument type that is not already implemented in the library.

To follow the code snippets used in this tutorial in Daml Studio, you can clone the Daml Finance repository and take a look at how the *Bond Instrument template* is implemented. In order to see how lifecycling is performed, you can run the script in the Instrument/Bond/Test/FixedRate.daml file.

Template Definition

We start by defining a new template for the instrument. Here are the fields used for the fixed rate instrument:

```
-- | This template models a fixed rate bond.
-- It pays a fixed coupon rate at the end of every coupon period.
template Instrument
  with
    depository : Party
      -- ^ The depository of the instrument.
    issuer : Party
      -- ^ The issuer of the instrument.
    id : Id
      -- ^ The identifier of the instrument.
    version : Text
      -- ^ The instrument's version.
    description : Text
      -- ^ A description of the instrument.
    couponRate : Decimal
      -- ^ The fixed coupon rate, per annum. For example, in case of a "3.5% p.a
↪coupon" this should
      --   be 0.035.
    periodicSchedule : PeriodicSchedule
      -- ^ The schedule for the periodic coupon payments.
    holidayCalendarIds : [Text]
      -- ^ The identifiers of the holiday calendars to be used for the coupon
↪schedule.
    calendarDataProvider : Party
      -- ^ The reference data provider to use for the holiday calendar.
    dayCountConvention : DayCountConventionEnum
      -- ^ The day count convention used to calculate day count fractions. For
↪example: Act360.
    currency : InstrumentKey
      -- ^ The currency of the bond. For example, if the bond pays in USD this
↪should be a USD cash
      --   instrument.
    notional : Decimal
      -- ^ The notional of the bond. This is the face value corresponding to one
↪unit of the bond
      --   instrument. For example, if one bond unit corresponds to 1000 USD,
↪this should be 1000.0.
    observers : PartiesMap
      -- ^ The observers of the instrument.
    lastEventTimestamp : Time
      -- ^ (Market) time of the last recorded lifecycle event. If no event has
↪occurred yet, the
      --   time of creation should be used.
```

These template variables describe the economic terms of a fixed rate bond.

The `Claims` Interface

We now need to map the template variables to a *Contingent Claims* tree, the internal representation we wish to use for lifecycling. Note that the Contingent Claims tree is not a part of the template above, instead it will be created dynamically upon request.

In order do that, we implement the *Claims interface*. This interface provides access to a generic mechanism to process coupon payments and redemptions. It will work in a similar way for the majority of instrument types, regardless of their specific economic terms.

Here is a high level implementation of the *Claims interface*:

```
    interface instance Claim.I for Instrument where
      view = Claim.View with acquisitionTime = dateToDateClockTime $
↪daysSinceEpochToDate 0
      getClaims Claim.GetClaims{actor} = do
        -- get the initial claims tree (as of the bond's acquisition time)
        let getCalendars = getHolidayCalendars actor calendarDataProvider
        (schedule, _) <- rollSchedule getCalendars periodicSchedule
↪holidayCalendarIds
        let
          useAdjustedDatesForDcf = True
          ownerReceives = True
          fxAdjustment = 1.0
          couponClaims = createFixRatePaymentClaims dateToDateClockTime schedule
↪periodicSchedule
            useAdjustedDatesForDcf couponRate ownerReceives dayCountConvention
↪notional currency
          redemptionClaim = createFxAdjustedPrincipalClaim dateToDateClockTime
↪ownerReceives
            fxAdjustment notional currency periodicSchedule.terminationDate
        pure [couponClaims, redemptionClaim]
```

The `getClaims` function is where we define the payoff of the instrument.

> First, we calculate the coupon payment dates by rolling out a periodic coupon schedule.
> The payment dates are then used to build claim sub-tree for the coupon payments.
> A claim sub-tree for the final redemption is also created.
> Finally, the coupon and redemption sub-trees are joined. Together, they yield the desired economic terms for the bond.

How to define the redemption claim

The the redemption claim depends on the currency and the maturity date of the bond.

```
-- | Create an FX adjusted principal claim.
-- This can be used for both FX swaps (using the appropriate FX rate) and single
↪currency bonds
-- (setting the FX rate to 1.0).
createFxAdjustedPrincipalClaim : (Date -> Time) -> Bool -> Decimal -> Decimal ->
↪Deliverable ->
  Date -> TaggedClaim
createFxAdjustedPrincipalClaim dateToTime ownerReceives fxRateMultiplier notional
  cashInstrument valueDate =
  let
```

(continues on next page)

(continued from previous page)

```
    fxLegClaimAmount = when (TimeGte valueDate)
                       $ scale (Const fxRateMultiplier)
                       $ scale (Const notional)
                       $ one cashInstrument
    fxLegClaim = if ownerReceives then fxLegClaimAmount else give fxLegClaimAmount
  in
    prepareAndTagClaims dateToTime [fxLegClaim] "Principal payment"
```

Keywords like *when*, *scale*, *one* and *give* should be familiar from the *Payoff Modeling tutorial*. *TimeGte* is just a synonym for *at*.

The `ownerReceives` flag is used to indicate whether the owner of a holding on the bond is meant to receive the redemption payment. When this is set to `false`, the holding custodian will be entitled to the payment.

How to define the coupon claims

The coupon claims are a bit more complicated to define. We need to take a schedule of adjusted coupon dates and the day count convention into account.

```
createFixRatePaymentClaimsList : Schedule -> PeriodicSchedule -> Bool -> Decimal -
↪> Bool ->
  DayCountConventionEnum -> Decimal -> Deliverable -> [C]
createFixRatePaymentClaimsList schedule periodicSchedule useAdjustedDatesForDcf
↪couponRate
  ownerReceives dayCountConvention notional cashInstrument =
    let
      couponDatesAdjusted = map (.adjustedEndDate) schedule
      couponAmounts = map (\p ->
          couponRate *
          (calcPeriodDcf dayCountConvention p useAdjustedDatesForDcf
            periodicSchedule.terminationDate periodicSchedule.frequency)
        ) schedule
      couponClaimAmounts = andList $
        zipWith
          (\d a ->
            when (TimeGte d) $ scale (Const a) $ scale (Const notional) $ one
↪cashInstrument
          ) couponDatesAdjusted couponAmounts
    in
      [if ownerReceives then couponClaimAmounts else give couponClaimAmounts]
```

For each coupon period, we calculate the adjusted end date and the actual coupon amount. We then create each coupon claim in a way similar to the redemption claim above.

Evolving the Instrument over time

The bond instrument gives the holder the right to receive future coupons and the redemption amount. At issuance, all coupons are due. However, after the first coupon is paid, the holder of the instrument is no longer entitled to receive it again. the `lastEventTimestamp` field in our template is used to keep track of the latest executed coupon payment.

Evolution of the instrument over time (and calculation of the corresponding lifecycle effects) can be performed using the *Lifecycle.Rule* template provided in the *Daml.Finance.Claims* package. This rule is very generic and can be used for all instruments that implement the *Claims interface*.

Let us break its implementation apart to describe what happens in more detail:

First, we retrieve the claim tree corresponding to the initial state of the instrument. We do so by fetching the `Claims` interface we defined for the template.

```
claimInstrument <- fetchInterfaceByKey @BaseInstrument.R instrument
```

By using the `lastEventTimestamp` (in our case: the last time a coupon was paid), we can now fast forward the claim tree to the current instrument state.

```
-- Recover claims tree as of the lastEventTimestamp. For a bond, this just requires
-- lifecycling as of the lastEventTimestamp.
dynInstrView <- BaseInstrument.exerciseInterfaceByKey @DynamicInstrument.I instrument lifecycler
  DynamicInstrument.GetView with viewer = lifecycler
let
  prevElections = map (\e ->
    case e.election of
      None -> error "election missing"
      Some (electorIsOwner, tag) -> electionEvent e.t electorIsOwner tag)
    dynInstrView.prevElections
  prevEvents = prevElections <> [timeEvent dynInstrView.lastEventTimestamp]

-- fast-forward the claims tree to the current version by replaying the previous events
-- (the previous elections + the lastEventTimestamp)
claims <- fst <$> lifecycle lifecycler observableCids claimInstrument prevEvents
```

Finally, we lifecycle the current instrument state to calculate pending cashflows. If there are such cashflows (for example when a coupon payment is due), we create a *Lifecycle Effect* for it, which can then be claimed and settled.

```
    evolve Lifecycle.Evolve{eventCid; observableCids; instrument} = do
      v <- view <$> fetch eventCid

      -- Fast-forward the instrument from inception to the timestamp of the last event.
      -- Then, perform a time-based lifecycling according to the current event.
      (remaining, pending, claims, claimInstrument, dynInstrView) <-
        fastForwardAndLifecycle instrument observableCids v.eventTime lifecycler

      let
```

(continues on next page)

(continued from previous page)

```
          pendingAfterNetting = netOnTag pending
          (otherConsumed, otherProduced) = splitPending pendingAfterNetting
        if remaining == claims && null pendingAfterNetting then
          pure (None, [])
        else do
          let
            currentKey = BaseInstrument.getKey $ toInterface claimInstrument
            newKey = currentKey with version = sha256 $ mconcat [show v.
↪eventTime, show remaining]
            producedInstrument = if isZero' remaining then None else Some
↪newKey
          tryCreateNewInstrument lifecycler dynInstrView.prevElections v.
↪eventTime None instrument
            newKey
          effectCid <- toInterfaceContractId <$> create Effect with
            providers = singleton currentKey.issuer
            id = v.id
            description = v.description
            targetInstrument = currentKey
            producedInstrument
            otherConsumed
            otherProduced
            settlementTime = Some v.eventTime
            observers = (.observers) . view $ toInterface @Disclosure.I
↪claimInstrument
          pure (Some newKey, [effectCid])
```

Including market observables

In our fixed rate bond example above, the coupon amount is pre-determined at the inception of the instrument. In contrast, a floating rate coupon is defined by the value of a reference rate during the lifetime of the bond. Since we do not know this value when the instrument is created, we need to define the coupon based on a future observation of the reference rate.

In the instrument definition, we need an identifier for the reference rate:

```
-- | This template models a floating rate bond.
-- It pays a floating coupon rate at the end of every coupon period.
-- This consists of a reference rate (observed at the beginning of the coupon
↪period) plus a coupon
-- spread. For example, 3M Euribor + 0.5%.
template Instrument
  with
    depository : Party
      -- ^ The depository of the instrument.
    issuer : Party
      -- ^ The issuer of the instrument.
    id : Id
      -- ^ An identifier of the instrument.
    version : Text
      -- ^ The instrument's version.
    description : Text
      -- ^ A description of the instrument.
    referenceRateId : Text
```

(continues on next page)

(continued from previous page)

```
      -- ^ The floating rate reference ID. For example, in case of "3M Euribor +
↪0.5%" this should
      --   be a valid reference to the "3M Euribor" reference rate.
```

When we create the claims for the coupon payments, we can then use *ObserveAt* to refer to the value of the reference rate:

```
-- | Calculate a floating rate amount for each payment date and create claims.
-- The floating rate is always observed on the first day of each payment period
↪and used for the
-- corresponding payment on the last day of that payment period. This means that
↪the calculation
-- agent needs to provide such an Observable, irrespective of the kind of
↪reference rate used (e.g.
-- a forward looking LIBOR or a backward looking SOFR-COMPOUND).
createFloatingRatePaymentClaims : (Date -> Time) -> Schedule -> PeriodicSchedule -
↪> Bool ->
  Decimal -> Bool -> DayCountConventionEnum -> Decimal -> Deliverable ->
↪Observable -> TaggedClaim
createFloatingRatePaymentClaims dateToTime schedule periodicSchedule
↪useAdjustedDatesForDcf
  floatingRateSpread ownerReceives dayCountConvention notional cashInstrument
↪referenceRateId =
  let
    couponClaimAmounts = andList $ map (\p ->
        when (TimeGte p.adjustedEndDate)
        $ scale (
          (ObserveAt referenceRateId p.adjustedStartDate + Const
↪floatingRateSpread) *
          (Const (calcPeriodDcf dayCountConvention p useAdjustedDatesForDcf
                    periodicSchedule.terminationDate periodicSchedule.frequency)
          ))
        $ scale (Const notional)
        $ one cashInstrument
      ) schedule
    couponClaims = if ownerReceives then couponClaimAmounts else give
↪couponClaimAmounts
  in prepareAndTagClaims dateToTime [couponClaims] "Floating rate payment"
```

In this example, the observable is a reference interest rate. Other instrument types can require other types of observables, such as an FX rate or a stock price.

Different ways to create and store the Contingent Claims tree

To summarize what we have seen so far, there are two different ways of using *Contingent Claims* in a Daml Finance instrument.

When using the *Generic Instrument*, we create the claim tree at instrument inception and store this representation explicitly on the ledger. After a lifecycle event, for example a coupon payment, a new version of the instrument (with a different claim tree) supersedes the previous version.

In contrast, the approach used in this tutorial only stores the key economic terms of the bond on the ledger. The claim tree is not stored on-ledger, but it is created on-the-fly when needed (for example, when lifecycling).

Which approach is preferred?

The latter approach has the advantage that the claim tree can adapt to changes in reference data. A change to e.g. a holiday calendar would automatically impact the claim tree the next time it is dynamically created. This is not the case for the first approach, where the tree is static.

Also, if the economic terms of the instrument result in a very large claim tree, it could be desirable not to store it on the ledger for performance reasons.

Finally, the dynamic approach allows for the terms of the template to be very descriptive to anyone familiar with the payoff at hand.

On the other hand, if you need to quickly create a one-off instrument, the on-ledger approach allows you to create the claims directly from a script, without first having to define a dedicated template.

Also, if you need to explicitly access Contingent Claims representations of older versions of the instrument on the ledger, for example for auditing reasons, that would be achieved out of the box with the first approach.

1.22 Reference

This reference section contains code patterns, a glossary, as well as code-level documentation for each Daml Finance package.

1.22.1 Glossary

This page defines some of the terminology used in the Daml Finance library.

We strive to use descriptive names and stay as close as possible to the traditional financial meaning of familiar terms.

1.22.1.1 Account

An account contract is a relationship between two parties: a custodian (or account provider) and an owner.

An account is referenced by *holdings* and it is used to control who is entitled to instruct and receive holding transfers.

1.22.1.2 Instrument

An instrument describes the economic terms (rights and obligations) of one unit of a financial contract.

An instrument is referenced by *holdings*. It can be as simple as an ISIN code referencing real-world (off-ledger) security, or it can encode specific on-ledger lifecycling logic.

1.22.1.3 Holding

A holding contract represents the ownership of a certain amount of an *Instrument* by an owner at a custodian.

1.22.1.4 Fungibility

Fungibility refers to the ability of an *Instrument* to be interchanged with other individual instruments of the same type. Fungible holdings can be `split` and `merged`.

1.22.1.5 Transferability

Transferability refers to the ability to transfer ownership of units of an *Instrument* to a new owner at the same custodian.

1.22.1.6 Locking

Locking is a mechanism that adds a third-party authorization requirement to any interaction with a *Holding* (archive, transfer, split, merge, etc.).

It is used to ensure that holdings committed to a certain workflow are not consumed by other workflows.

1.22.1.7 Crediting / Debiting

Crediting is the process of creating new *Holdings* for a given instrument and debiting, conversely, is removing existing ones.

1.22.1.8 Disclosure

Disclosure is the ability to reveal a contract to a third party by adding them as an observer.

1.22.1.9 Settlement

Settlement is the (possibly simultaneous) execution of ownership transfers according to predefined instructions.

Many financial transactions are traditionally settled a few days after execution.

1.22.1.10 Lifecycling

Lifecycling refers to the evolution of *Instruments* over their lifetime.

Lifecycling can deal with intrinsic events, like contractual cash flows, and/or extrinsic events like corporate actions or elections.

1.22.2 Patterns

This page explains some common design patterns used in the Daml Finance library.

1.22.2.1 Factory pattern

Factories are helper contracts that are used to create instruments, holdings, and other contracts. The reason why using factories is a recommended pattern when using Daml Finance has to do with application decoupling / upgradeability of your application.

For example, suppose that you are writing Daml code to issue equity instruments. Your workflow references the version `0.2.1` of the *Equity implementation package* and at some point creates an instrument as follows.

```
create Equity.Instrument with
  issuer = myParty
  id = Id "MyCompany"
  ..
```

If the equity package gets updated to version `0.2.2` and a new field is added to the instrument (or a choice is changed, or a new lifecycle event is added,) then you are forced to upgrade your Daml code in order to use the new feature and will have to deal with upgrading multiple templates on the ledger.

A safer approach is for your Daml code to only reference the *Equity interface package*, which contains interface definitions and is updated less frequently.

However, you would now need a way to create equity instruments without referencing `Daml.Finance.Instrument.Equity` in your main Daml workflow. To do this, you can setup a Script to run during ledger initialisation that will create a *factory contract* and cast it to the corresponding *interface*. You can then use the factory in your main workflow code to create the instruments.

When an upgraded instrument comes along, you would need to write code to archive the old factory and create the new one, in order to issue the new instruments. However, the Daml code for your workflow could in principle stay untouched.

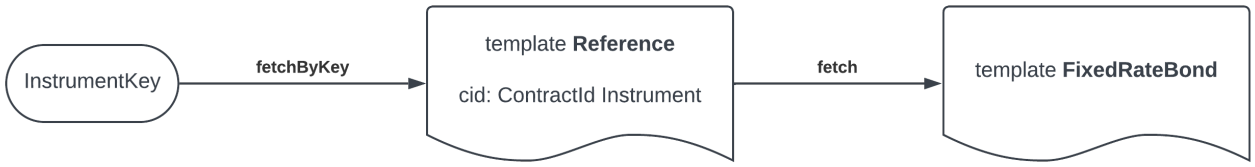
For an example where the Factory pattern is used, check out the *Holdings tutorial*.

1.22.2.2 Reference pattern

The Reference pattern is used to leverage the functionalities of *Contract Keys* when working with interfaces. This is required as there is currently no built-in support at the language level for interface keys.

We want for instance to use an *InstrumentKey* to identify instruments across a number of implementing templates.

To do that, we define a *Reference* template that
is keyed by the *InstrumentKey*
contains the interface contract id of the target instrument
We can then fetch an interface by key by
fetching the *Reference* template by key (`fetchByKey`)
reading and `fetch`-ing the stored contract id



Similarly, this pattern also lets us exercise a choice on an interface by key .

The Reference pattern is currently used in Daml Finance for instruments and accounts, where we ensure that a template and its companion *Reference* template are kept in sync. It is however important to understand this pattern should you implement custom instruments or accounts.

1.22.2.3 View of an interface contract and the *GetView* choice

There are different ways to access the data of a contract, for example the terms of an instrument:

1. *fetch* the interface contract using its contract ID (this requires the submitting party to be a stakeholder of the contract). It is then possible to use the `view` built-in method to get the interface view.
2. `GetView`: by calling this choice on the interface, for example on a *callable bond*, a party can get the view of a contract, without necessarily being a stakeholder of the contract. This can be useful in situations where someone needs access to reference data, but should not be a stakeholder of the contract. Specifically, if *publicParty* is an observer of an instrumentCid, a party would only require readAs rights of *publicParty* in order to exercise `GetView`. In the Daml Finance library, this choice has been implemented not only for instruments but also for other types of contracts, e.g. *Holdings* and lifecycle related contracts like *Rule* and *Effect*.

1.22.3 Daml Finance

Here is a complete list of modules in the financial library:

1.22.3.1 ContingentClaims.Core.Builders

Functions

unrollDates : Int -> Int -> [Month] -> Int -> [Date]
Helper function to generate a series of fixing dates, e.g. for coupon payments in `fixed`. This assumes `fixingMonths` and `fixingDates` are ordered. The Daml Finance library(https://github.com/digital-asset/daml-finance) has more feature-complete date handling functions.

forward : t -> *Observation* t x o -> *Claim* t x a o -> *Claim* t x a o
Forward agreement. Discounted by (potentially stochastic) interest rate `r`.

fra : t -> t -> *Observation* t x o -> *Observation* t x o -> *Claim* t x a o -> *Claim* t x a o
Forward rate agreement.

zcb : t -> x -> ccy -> *Claim* t x ccy o
Zero Coupon Bond.

floating : *Observation* t x o -> *Observation* t x o -> ccy -> [t] -> *Claim* t x ccy o
A floating rate bond.

fixed : x -> x -> ccy -> [t] -> *Claim* t x ccy o
A (fixed rate) coupon paying bond.

european : t -> *Claim* t x a o -> *Claim* t x a o
European option on the input claim. At maturity, the holder must `EXERCISE` or `EXPIRE` the claim. e.g. call option on S&P 500:

```
european (date 2021 05 14) (observe "SPX" - pure 4200)
```

bermudan : [t] -> *Claim* t x a o -> *Claim* t x a o
Bermudan option on the input claim. Given a pre-defined set of times {t_1, t_2, .., t_N}, it allows the holder to acquire the underlying claim on at most one of these times. At each election time before maturity, the holder must `EXERCISE` the option or `POSTPONE`. At maturity, the holder must `EXERCISE` or `EXPIRE`.

american : t -> t -> *Claim* t x a o -> *Claim* t x a o
American option (knock-in). The lead parameter is the first possible acquisition date.

swap : ([t] -> *Claim* t x a o) -> ([t] -> *Claim* t x a o) -> [t] -> *Claim* t x a o
Asset swap on specific fixing dates `[t]`. For example:

```
fixedUsdVsFloatingEur : [t] -> Serializable.Claim Text
fixedUsdVsFloatingEur =
  fixed 100.0 0.02 "USD" `swap` floating (observe "USDEUR" * pure 100.0)
↪(observe "EUR1M") "EUR"
```

1.22.3.2 ContingentClaims.Core.Claim

Functions

zero : *Claim* t x a o
Constructs a claim without rights or obligations.

one : a -> *Claim* t x a o
Constructs a claim that delivers one unit of `a` immediately to the bearer.

give : *Claim* t x a o -> *Claim* t x a o
Constructs a claim that reverses the obligations of the bearer and their counterparty.

and : *Claim* t x a o -> *Claim* t x a o -> *Claim* t x a o
Used to additively combine two claims together. In order to use this, you must import this module qualified or hide the `and` operator from `Prelude`.

or : *Electable* t x a o -> *Electable* t x a o -> *Claim* t x a o
Gives the bearer the right to choose between the input claims. In order to use this, you must import this module qualified or hide the `or` operator from `Prelude`.

andList : [*Claim* t x a o] -> *Claim* t x a o
Used to additively combine a list of claims together. It is equivalent to applying the `and` builder recursively.

orList : [*Electable* t x a o] -> *Claim* t x a o
Gives the bearer the right to choose between the input claims. It is equivalent to applying the `or` builder recursively.

cond : *Inequality* t x o -> *Claim* t x a o -> *Claim* t x a o -> *Claim* t x a o
Gives the bearer the right to the first claim if predicate is true, else the second claim.

scale : *Observation* t x o -> *Claim* t x a o -> *Claim* t x a o
Multiplies the input claim by a scaling factor (which can be non-deterministic).

when : *Inequality* t x o -> *Claim* t x a o -> *Claim* t x a o
Acquires the input claim on *the first instant* that `predicate` is true.

anytime : *Inequality* t x o -> Text -> *Claim* t x a o -> *Claim* t x a o
Gives the bearer the right to enter a claim at any time `predicate` is true.

until : *Inequality* t x o -> *Claim* t x a o -> *Claim* t x a o
Expires the input claim on the *first instant* that `predicate` is true.

mapParams : (t -> i) -> (i -> t) -> (a -> a') -> (o -> o') -> (x -> x') -> *Claim* i x a o -> *Claim* t x' a' o'
Replaces parameters in a claims using the input mapping functions. This can be used to e.g. map the time parameter in a claim from `Date` to `Time`, or to map the asset type parameter from an abstract `Text` to a concrete `InstrumentKey`.

at : t -> *Inequality* t x o
Given `t`, constructs a predicate that is `True` for time `t`, `False` otherwise.

upTo : t -> *Inequality* t x a
Given `t`, constructs a predicate that is `True` for time `t`, `False` otherwise.

(<=) : *Observation* t x o -> *Observation* t x o -> *Inequality* t x o
Given observations `o1` and `o2`, constructs the predicate `o1 ≤ o2`. In order to use this, you must import this module qualified or hide the `(<=)` operator from `Prelude`.

compare : (Ord t, Ord x, Number x, Divisible x, CanAbort m) => (o -> t -> m x) -> *Inequality* t x o -> t -> m Bool

Reify the `Inequality` into an observation function. This function is used to convert an abstract predicate, e.g. `S ≤ 50.0` to the actual boolean observation function `t -> m Bool`.

1.22.3.3 ContingentClaims.Core.Internal.Claim

Data Types

data *Claim* t x a o

Core data type used to model cashflows of instruments. In the reference paper from Peyton-Jones this is called 'Contract'. We renamed it to avoid ambiguity.

`t` corresponds to the time parameter.
`x` corresponds to the `Observation` output type. An observation is a function from `t` to `x`. A common choice is to use `Time` and `Decimal`, respectively.
`a` is the representation of a deliverable asset, e.g. a `Text` ISIN code or an `InstrumentKey`.
`o` is the representation of an observable, e.g. a `Text`.

You should build the `Claim` using the smart constructors (e.g. `zero`, `and`) instead of using the data constructors directly (`Zero`, `And`).

Zero

Represents an absence of claims. Monoid `And` identity.

One a

The bearer acquires one unit of `a` *immediately*.

Give (*Claim* t x a o)

The obligations of the bearer and the counterparty are reversed.

And

Used to combine multiple rights together.

Field	Type	Description
a1	*Claim* t x a o	
a2	*Claim* t x a o	
as	[*Claim* t x a o]	

Or

Gives the bearer the right to choose between several claims.

Field	Type	Description
or1	*Electable* t x a o	
or2	*Electable* t x a o	
ors	[*Electable* t x a o]	

Cond

Gives the bearer the right to the first claim if `predicate` is true, else the second claim.

Field	Type	Description
predicate	*Inequality* t x o	
success	*Claim* t x a o	
failure	*Claim* t x a o	

Scale

Multiplies the `claim` by `k` (which can be non-deterministic).

Field	Type	Description
k	*Observation* t x o	
claim	*Claim* t x a o	

When

Defers the acquisition of `claim` until *the first instant* that `predicate` is true.

Field	Type	Description
predicate	*Inequality* t x o	
claim	*Claim* t x a o	

Anytime

Gives the bearer the right to enter a claim at any time the predicate is true.

Field	Type	Description
predicate	*Inequality* t x o	
electable	*Electable* t x a o	

Until

Expires said claim on the *first instant* that `predicate` is true.

Field	Type	Description
predicate	*Inequality* t x o	
claim	*Claim* t x a o	

instance Corecursive (*Claim* t x a o) (ClaimF t x a o)

instance Recursive (*Claim* t x a o) (ClaimF t x a o)

instance (Eq a, Eq x, Eq o, Eq t) => Eq (*Claim* t x a o)

instance (Show t, Show x, Show a, Show o) => Show (*Claim* t x a o)

instance Monoid (*Claim* t x a o)

instance Semigroup (*Claim* t x a o)

type ***Electable*** **t x a o** = (Text, *Claim* t x a o)

Type synonym for sub-trees that can be elected in an `Or` or `Anytime` node. The textual tag is used to identify each sub-tree when an election is made.

data *Inequality* t x o

Data type for boolean predicates supported by the library. A boolean predicate is a generic function with signature `t -> Bool`. However, a limited set of predicates is currently supported.

TimeGte t

`True` when `time ≥ t`, `False` otherwise.

TimeLte t

`True` when `time ≤ t`, `False` otherwise.

Lte (*Observation* t x o, *Observation* t x o)

`True` when `o(t) ≤ o'(t)`, `False` otherwise, for a pair of observations `o`, `o'`.

instance (Eq t, Eq x, Eq o) => Eq (*Inequality* t x o)

instance (Show t, Show x, Show o) => Show (*Inequality* t x o)

1.22.3.4 ContingentClaims.Core.Observation

Data Types

data *Observation* t x o

Implementation of market observables. Conceptually it is helpful to think of this as the type `t -> x`, or `t -> Update x`.

Const

A numerical constant, e.g. `10.0`.

Field	Type	Description
value	x	

Observe

A named parameter, e.g. "LIBOR 3M".

Field	Type	Description
key	o	

ObserveAt

A named parameter, e.g. "LIBOR 3M", observed at an explicit point in time.

Field	Type	Description
key	o	
t	t	

Add (*Observation* t x o, *Observation* t x o)

Sum of two observations.

Neg (*Observation* t x o)

Opposite of an observation.

Mul (*Observation* t x o, *Observation* t x o)

Product of two observations.

Div (*Observation* t x o, *Observation* t x o)

Division of two observations.

instance Corecursive (*Observation* t x o) (ObservationF t x o)

instance Recursive (*Observation* t x o) (ObservationF t x o)

instance Functor (*Observation* t x)

instance (Eq x, Eq o, Eq t) => Eq (*Observation* t x o)

instance Additive x => Additive (*Observation* t x o)

instance Multiplicative x => Divisible (*Observation* t x o)

instance Multiplicative x => Multiplicative (*Observation* t x o)

instance (Additive x, Multiplicative x) => Number (*Observation* t x o)

instance (Show t, Show x, Show o) => Show (*Observation* t x o)

Functions

pure : x -> *Observation* t x o
Smart constructor for `Const`. Lifts a constant to an observation.

observe : o -> *Observation* t x o
Smart constructor for `Observe`. Looks up the value of `o`.

eval : (Ord t, Number x, Divisible x, CanAbort m) => (o -> t -> m x) -> *Observation* t x o -> t -> m x
Reify the `Observation` into an observation function. This function is used to convert an abstract observation, e.g. `LIBOR 3M + 0.005` to the actual observation function `t -> m x`.

mapParams : (i -> t) -> (o -> o') -> (x -> x') -> *Observation* i x o -> *Observation* t x' o'
The functor map operation *and* also map any parameters to keys. For example, could map the param "spot" to an ISIN code "GB123456789". Also contra-maps time parameter, i.e. from relative time values to absolute ones.
@ mapParams identity = bimap

1.22.3.5 ContingentClaims.Core.Util.Recursion

This module collects a set of utilities used to execute recursion schemes. The morphisms ending in 'M' are monadic variants, allowing to interleave, e.g., `Update` or `Script`. `cataM` after Tim Williams' talk, https://www.youtube.com/watch?v=Zw9KeP3OzpU.

Functions

paraM : (Monad m, Traversable f, Recursive b f) => (f (b, a) -> m a) -> b -> m a
Monadic paramorphism.

anaM : (Monad m, Traversable f, Corecursive b f) => (a -> m (f a)) -> a -> m b
Monadic anamorphism.

apoM : (Monad m, Traversable f, Corecursive b f) => (a -> m (f (Either b a))) -> a -> m b
Monadic apomorphism.

futuM : (Monad m, Traversable f, Corecursive b f) => (a -> m (f (Free f a))) -> a -> m b
Monadic futumorphism.

apoCataM : (Monad m, Traversable f, Corecursive b f) => (f b -> b) -> (a -> m (f (Either b a))) -> a -> m b
Monadic lazy unfold (apoM) followed by a fold (cata). This Specialised lazy re-fold is used by `lifecycle`.

hyloM : (Traversable f, Monad n) => (f b -> b) -> (a -> n (f a)) -> a -> n b
A modified `hylo` (refold), whith an interleaved monad effect (typically `Update`).

ghyloM : (Comonad w, Traversable f, Monad m, Traversable m, Monad n) => (f (w c) -> w (f c)) -> (m (f d) -> f (m d)) -> (f (w b) -> b) -> (a -> n (f (m a))) -> a -> n b
Generalised hylomorphism (with monadic unfold).

funzip : Functor f => f (a, b) -> (f a, f b)
Functor unzip.

synthesize : (Functor f, Recursive b f) => (f attr -> attr) -> b -> Cofree f attr
Annotate a recursive type bottom-up.

inherit : (Functor f, Corecursive b f, Recursive b f) => (b -> attr -> attr) -> attr -> b -> Cofree f attr
Annotate a recursive type top-down.

1.22.3.6 ContingentClaims.Lifecycle.Lifecycle

Data Types

data *Pending* t a

Used to specify pending payments.

Pending

Field	Type	Description
t	t	Payment time.
amount	Decimal	Amount of asset to be paid.
asset	a	Asset in which the payment is denominated.

instance (Eq t, Eq a) => Eq (*Pending* t a)

instance (Show t, Show a) => Show (*Pending* t a)

data *Result* t a o

Returned from a `lifecycle` operation.

Result

Field	Type	Description
pending	[*Pending* t a]	Payments requiring settlement.
remaining	C t a o	The tree after lifecycled branches have been pruned.

instance (Eq a, Eq o, Eq t) => Eq (*Result* t a o)

instance (Show t, Show a, Show o) => Show (*Result* t a o)

Functions

lifecycle : (Ord t, Eq a, CanAbort m) => (o -> t -> m Decimal) -> C t a o -> t -> t -> m (*Result* t a o)

Collect claims falling due into a list, and return the tree with those nodes pruned. `m` will typically be `Update`. It is parametrised so it can be run in a `Script`. The first argument is used to lookup the value of any `Observables`. Returns the pruned tree + pending settlements up to the provided market time.

exercise : (Ord t, Eq a, Eq o, CanAbort m) => (o -> t -> m Decimal) -> (Bool, Text) -> C t a o -> t -> t -> m (C t a o)

Acquire `Anytime` and `Or` nodes, by making an election. Import this `qualified` to avoid clashes with `Prelude.exercise`.

1.22.3.7 ContingentClaims.Lifecycle.Util

This module defines a set of utility functions to extract information from claim trees.

Functions

fixings : *Claim* t x a o -> [t]
Return the fixing dates of a claim. This does not discriminate between optional dates which *may* result from a condition, and outright fixings. It also does not correctly account for malformed trees, where subtrees are orphaned due to impossible `When` statements, e.g., `When (t > 1) ((When t < 1) _)`.

expiry : Ord t => *Claim* t x a o -> Optional t
Return the time after which the claim is worthless, i.e., value = 0, if such a time exists. Also known as 'maturity' or 'horizon' in the Eber/Jones paper.

payoffs : (Eq t, Eq x, Eq o, Multiplicative x) => *Claim* t x a o -> [(*Observation* t x o, a)]
Return a list of possible scale-factor/payoff pairs. This does not discriminate between conditional and outright payoffs.

pruneZeros : *Claim* t x a o -> *Claim* t x a o
Prunes sub-trees which are `Zero`.

isZero : *Claim* t x a o -> Bool
Checks if a claim is the `Zero` claim. This avoids requiring the equality type constraint on `a`.

1.22.3.8 ContingentClaims.Valuation.MathML

Typeclasses

class *ToXml* a **where**

Renders an `Expr` into MathML presentation format.

presentation : a -> Xml

instance *ToXml* t => *ToXml* (*Expr* t)

instance *ToXml* Decimal

instance *ToXml* Text

instance *ToXml* Date

1.22.3.9 ContingentClaims.Valuation.Stochastic

Typeclasses

class *IsIdentifier* t **where**

localVar : Int -> t
Produce a local identifier of type `t`, subindexed by `i`.

Data Types

data *Expr* t

Represents an expression of t-adapted stochastic processes.

Const Decimal

Ident t

Proc

Field	Type	Description
name	Text	
process	*Process* t	
filtration	t	

Sup

Field	Type	Description
lowerBound	t	
tau	t	
rv	*Expr* t	

Sum [*Expr* t]

Neg (*Expr* t)

Mul (*Expr* t, *Expr* t)

Pow (*Expr* t, *Expr* t)

I (*Expr* t, *Expr* t)

E

Field	Type	Description
rv	*Expr* t	
filtration	t	

instance *ToXml* t => *ToXml* (*Expr* t)

instance Corecursive (*Expr* t) (*ExprF* t)

instance Recursive (*Expr* t) (*ExprF* t)

instance Eq t => Eq (*Expr* t)

instance Show t => Show (*Expr* t)

data *ExprF* t x

Base functor for `Expr`. Note that this is ADT is re-used in a couple of places, e.g., `Process`, where however not every choice is legal and will lead to a partial evaluator.

ConstF Decimal

IdentF t

ProcF

Field	Type	Description
name	Text	
process	*Process* t	
filtration	t	

SupF

Field	Type	Description
lowerBound	t	
tau	t	
rv	x	

SumF [x]

NegF x

MulF

Field	Type	Description
lhs	x	
rhs	x	

PowF

Field	Type	Description
lhs	x	
rhs	x	

I_F

Field	Type	Description
lhs	x	
rhs	x	

E_F

Field	Type	Description
rv	x	
filtration	t	

instance Corecursive (*Expr* t) (*ExprF* t)

instance Recursive (*Expr* t) (*ExprF* t)

instance Functor (*ExprF* t)

instance Foldable (*ExprF* t)

instance Traversable (*ExprF* t)

data *Process* t

A stochastic processes. Currently this represents a Geometric Browniam Motion, i.e., dX / X = dt + dW. Eventually, we wish to support other processes such as Levy.

Process

Field	Type	Description
dt	*Expr* t	
dW	*Expr* t	

instance Eq t => Eq (*Process* t)

instance Show t => Show (*Process* t)

Functions

riskless : t -> *Process* t

Helper function to create a riskless process `dS = r dt`.

gbm : t -> t -> *Process* t

Helper function to create a geometric BM `dS = μ dt + σ dW`.

fapf : (Eq a, Show a, Show o, *IsIdentifier* t) => a -> (a -> *Process* t) -> (a -> a -> *Process* t) -> (o -> *Process* t) -> t -> *Claim* t Decimal a o -> *Expr* t

Converts a `Claim` into the Fundamental Asset Pricing Formula. The expressions are defined as E1-E10 in the Eber/Peyton-Jones paper. This is still an experimental feature.

simplify : *Expr* t -> *Expr* t

This is meant to be a function that algebraically simplifies the FAPF by

1. using simple identities and ring laws
2. change of numeraire technique. This is still an experimental feature.

1.22.3.10 Daml.Finance.Account.Account

Templates

template *Account*

A relationship between a custodian and an asset owner. It is referenced by holdings.

Signatory: custodian, owner

Field	Type	Description
custodian	Party	The account provider.
owner	Party	The account owner.
controllers	*Controllers*	Controllers of transfers.
id	*Id*	Identifier of the account.
description	Text	Description of the account.
holdingFactoryCid	ContractId *F*	Associated holding factory.
observers	*PartiesMap*	Observers.

Choice Archive
Controller: custodian, owner
Returns: ()
(no fields)
interface instance *I* **for** *Account*
interface instance *I* **for** *Account*

template *Factory*

Template used to create accounts.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *F* **for** *Factory*
interface instance *I* **for** *Factory*

Data Types

type *T* = *Account*
Type synonym for `Account`.

1.22.3.11 Daml.Finance.Claims.Lifecycle.Rule

Templates

template *Rule*

Rule to process an event for instruments that are modelled using "on-the-fly" claims (the tree is not stored on-ledger but generated dynamically). This rule supports both time update events and election events.

Signatory: providers

Field	Type	Description
providers	*Parties*	Providers of the lifecycling rule.
lifecycler	Party	Party performing the lifecycling.
observers	*PartiesMap*	Observers of the rule.
id	*Id*	Identifier for the rule contract.
description	Text	Textual description.

Choice Archive
Controller: providers
Returns: ()
(no fields)
interface instance *Exercisable* **for** *Rule*
interface instance *I* **for** *Rule*
interface instance *I* **for** *Rule*

1.22.3.12 Daml.Finance.Claims.Util

Functions

isZero : Party -> *I* -> Update Bool
Checks if all input claims are zero.

isZero' : [*TaggedClaim*] -> Bool
Checks if all input claims are zero.

toTime : (t -> Time) -> *Claim* t x a o -> *Claim* Time x a o
Maps the time parameter in a `Claim` to `Time`. As `Time` is generally understood to express UTC time, we recommend mapping to UTC time.

1.22.3.13 Daml.Finance.Claims.Util.Builders

This module includes utility functions used to build contingent claim trees that represent specific payoffs. A `Schedule` is usually used as an input to these utility functions. Given that schedules are defined in terms of dates, a claim where the time parameter is `Date` is returned. These are then mapped to claims where the time parameter is `Time` using a (user-provided) conversion function.

Data Types

type ***C*** = *Claim* Date Decimal *Deliverable Observable*

type ***O*** = *Observation* Date Decimal *Observable*

Functions

prepareAndTagClaims : (Date -> Time) -> [*C*] -> Text -> *TaggedClaim*
Convert the claims to UTCTime and tag them.

createFixRatePaymentClaimsList : *Schedule* -> *PeriodicSchedule* -> Bool -> Decimal -> Bool -> *DayCountConventionEnum* -> Decimal -> *Deliverable* -> [*C*]

createFixRatePaymentClaims : (Date -> Time) -> *Schedule* -> *PeriodicSchedule* -> Bool -> Decimal -> Bool -> *DayCountConventionEnum* -> Decimal -> *Deliverable* -> *TaggedClaim*
Calculate a fix rate amount for each payment date and create claims.

createConditionalCreditFixRatePaymentClaims : (Date -> Time) -> *Schedule* -> *PeriodicSchedule* -> Bool -> Decimal -> Bool -> *DayCountConventionEnum* -> Decimal -> *Deliverable* -> *Observable* -> *TaggedClaim*
Calculate a fix rate amount (if a credit event has not yet happened) for each payment date and create claims.

createCreditEventPaymentClaims : (Date -> Time) -> Bool -> Decimal -> *Deliverable* -> *Observable* -> *Observable* -> *PeriodicSchedule* -> *TaggedClaim*
Calculate a (1-recoveryRate) payment if a credit event just happened and create claims.

createFloatingRatePaymentClaims : (Date -> Time) -> *Schedule* -> *PeriodicSchedule* -> Bool -> Decimal -> Bool -> *DayCountConventionEnum* -> Decimal -> *Deliverable* -> *Observable* -> *TaggedClaim*
Calculate a floating rate amount for each payment date and create claims. The floating rate is always observed on the first day of each payment period and used for the corresponding payment on the last day of that payment period. This means that the calculation agent needs to provide such an Observable, irrespective of the kind of reference rate used (e.g. a forward looking LIBOR or a backward looking SOFR-COMPOUND).

createAssetPerformancePaymentClaims : (Date -> Time) -> *Schedule* -> *PeriodicSchedule* -> Bool -> Bool -> *DayCountConventionEnum* -> Decimal -> *Deliverable* -> *Observable* -> *TaggedClaim*
Calculate the asset performance for each payment date and create claims. The performance is calculated using the reference asset from the start date to the end date of each payment period. The reference asset Observable needs to contain the appropriate type of fixings:
unadjusted fixings in case of a price return asset swap
adjusted fixings in case of a total return asset swap

createFxAdjustedPrincipalClaim : (Date -> Time) -> Bool -> Decimal -> Decimal -> *Deliverable* -> Date -> *TaggedClaim*

Create an FX adjusted principal claim. This can be used for both FX swaps (using the appropriate FX rate) and single currency bonds (setting the FX rate to 1.0).

createVanillaOptionClaim : (Date -> Time) -> Decimal -> *Observable* -> *Deliverable* -> Date -> Bool -> *C*
Create the claim for a long vanilla option (cash-settled, automatically exercised).

createEuropeanCashClaim : (Date -> Time) -> Bool -> Decimal -> *Observable* -> *Deliverable* -> Date -> Bool -> *TaggedClaim*
Create the claim for a cash-settled, automatically exercised option (long or short).

createBarrierEuropeanCashClaim : (Date -> Time) -> Bool -> Decimal -> *Observable* -> *Deliverable* -> Date -> Bool -> Decimal -> Date -> Bool -> Bool -> *TaggedClaim*
Create the claim for a barrier option (automatically exercised, cash-settled).

createEuropeanPhysicalClaim : (Date -> Time) -> Bool -> Decimal -> *Deliverable* -> *Deliverable* -> Date -> Bool -> *TaggedClaim*
Create the claim for a physically settled European option.

createDividendOptionClaim : (Date -> Time) -> Date -> *InstrumentQuantity* -> Optional *InstrumentQuantity* -> Optional *InstrumentQuantity* -> *TaggedClaim*
Create the claim for a physically settled Dividend option.

1.22.3.14 Daml.Finance.Claims.Util.Date

Data Types

type *O* = *Observation* Date Decimal *Observable*

Functions

convertImplicitDcfToActualDcf : *O* -> *SchedulePeriod* -> Bool -> *PeriodicSchedule* -> *DayCountConventionEnum* -> *O*
Calculate a conversion factor if the dcf used for a floating rate compounded index does not match the dcf used for an instrument.

1.22.3.15 Daml.Finance.Claims.Util.Lifecycle

Functions

timeEvent : Time -> EventData
Constructor for a time event.

electionEvent : Time -> Bool -> Text -> EventData
Constructor for an election event.

lifecycleClaims : [ContractId *I*] -> Time -> [*TaggedClaim*] -> [EventData] -> Update ([*TaggedClaim*], [*Pending*])
Lifecycle a set of claims at specified events.

netOnTag : [*Pending*] -> [*Pending*]
Net pending payments on the same instrument, which also have the same tag.

lifecycle : Party -> [ContractId *I*] -> *I* -> [EventData] -> Update ([*TaggedClaim*], [*Pending*])
Lifecycle a claim instrument at specified events.

splitPending : [*Pending*] -> ([*InstrumentQuantity*], [*InstrumentQuantity*])
Map pending settlements into corresponding instrument quantities and split them into consumed and produced. Pending items with an amount of `0.0` are discarded.

1.22.3.16 Daml.Finance.Data.Numeric.Observation

Templates

template *Factory*

Implementation of the corresponding Observation Factory.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *F* **for** *Factory*
interface instance *I* **for** *Factory*

template *Observation*

An implementation of `NumericObservable` that explicitly stores time-dependent numerical values. For example, it can be used for equity or rate fixings.

Signatory: provider

Field	Type	Description
provider	Party	The reference data provider.
id	*Id*	A textual identifier.
observations	Map Time Decimal	The time-dependent values.
observers	*PartiesMap*	Observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *I* **for** *Observation*
interface instance *I* **for** *Observation*
interface instance *I* **for** *Observation*

Data Types

type *T* = *Observation*
Type synonym for `Observation`.

1.22.3.17 Daml.Finance.Data.Reference.HolidayCalendar

Templates

template *Factory*

Implementation of the corresponding HolidayCalendar Factory.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *F* **for** *Factory*
interface instance *I* **for** *Factory*

template *HolidayCalendar*

Holiday calendar of an entity (typically an exchange or a currency). It is maintained by a reference data provider.

Signatory: provider

Field	Type	Description
calendar	*HolidayCalendarData*	Holiday Calendar Data used to define holidays.
observers	*PartiesMap*	Observers.
provider	Party	The party maintaining the `HolidayCalendar`.

Choice Archive
Controller: provider
Returns: ()
(no fields)
Choice *GetCalendar*
Returns the calendar's `HolidayCalendarData`.
Controller: viewer
Returns: *HolidayCalendarData*

Field	Type	Description
viewer	Party	The party fetching the calendar.

interface instance *I* **for** *HolidayCalendar*

interface instance *I* **for** *HolidayCalendar*

Data Types

data *HolidayCalendarKey*

Key used to look up the holiday calendar of an entity, as defined by a reference data provider.

HolidayCalendarKey

Field	Type	Description
provider	Party	The party maintaining the `HolidayCalendar`.
id	Text	A textual label identifying the calendar (e.g. "NYSE" for the New York Stock Exchange holiday calendar).

instance Eq *HolidayCalendarKey*

instance Show *HolidayCalendarKey*

instance HasExerciseByKey *HolidayCalendar HolidayCalendarKey GetCalendar HolidayCalendarData*

instance HasExerciseByKey *HolidayCalendar HolidayCalendarKey* Archive ()

instance HasFetchByKey *HolidayCalendar HolidayCalendarKey*

instance HasFromAnyContractKey *HolidayCalendar HolidayCalendarKey*

instance HasKey *HolidayCalendar HolidayCalendarKey*

instance HasLookupByKey *HolidayCalendar HolidayCalendarKey*

instance HasMaintainer *HolidayCalendar HolidayCalendarKey*

instance HasToAnyContractKey *HolidayCalendar HolidayCalendarKey*

Functions

getHolidayCalendars : Party -> Party -> [Text] -> Update [*HolidayCalendarData*]
Retrieve holiday calendar(s) from the ledger.

rollSchedule : ([Text] -> Update [*HolidayCalendarData*]) -> *PeriodicSchedule* -> [Text] -> Update (*Schedule*, [*HolidayCalendarData*])
Retrieve holiday calendar(s) from the ledger and roll out a schedule. Returns the rolled schedule and the required calendars.

1.22.3.18 Daml.Finance.Data.Time.DateClock

Templates

template *DateClock*

A `DateClock` is a template used to keep track of the current date. It implements the `Time` rule interface to be able to advance and rewind business time. It also implements the `TimeObservable` interface. Specifically, each date `D` is mapped to `D 00:00:00 UTC`. If your use-case involves working across multiple time zones, you may need to define multiple `DateClock` templates with specific time conversions.

Signatory: providers

Field	Type	Description
providers	*Parties*	The clock's providers.
date	*Unit*	The clock's date.
id	*Id*	The clock's identifier.
description	Text	The clock's description.
observers	*Parties*	Observers.

Choice Archive
Controller: providers
Returns: ()
(no fields)
interface instance *I* **for** *DateClock*
interface instance *I* **for** *DateClock*

Data Types

type *T* = *DateClock*
Type synonym for `DateClock`.

Functions

dateToDateClockTime : Date -> Time
Maps a `Date` to `Time` using the rule in the `DateClock`.

1.22.3.19 Daml.Finance.Data.Time.DateClock.Types

Data Types

data *Unit*

A `Date` which can be converted to `Time`. Specifically, each date `D` is mapped to `D 00:00:00 UTC`.

Unit Date

instance *HasUTCTimeConversion Unit*

instance Eq *Unit*

instance Ord *Unit*

instance Show *Unit*

1.22.3.20 Daml.Finance.Data.Time.DateClockUpdate

Templates

template *DateClockUpdateEvent*

Event signalling the update of a `DateClock`. It can trigger the execution of lifecycle rules for some instruments.

Signatory: providers

Field	Type	Description
providers	*Parties*	Providers of the event.
date	Date	The updated clock data.
eventTime	Time	The event time.
id	*Id*	Event identifier.
description	Text	Event description.
observers	*Parties*	The clock's observers.

Choice Archive
Controller: providers
Returns: ()
(no fields)
interface instance *I* **for** *DateClockUpdateEvent*
interface instance *I* **for** *DateClockUpdateEvent*

Data Types

type *T* = *DateClockUpdateEvent*
Type synonym for `DateClockUpdateEvent`.

1.22.3.21 Daml.Finance.Data.Time.LedgerTime

Templates

template *LedgerTime*

A `LedgerTime` is a template used to retrieve current ledger time as a `TimeObservable`.

Signatory: providers

Field	Type	Description
providers	*Parties*	The time providers.
id	*Id*	The ledger time identifier.
description	Text	The ledger time description.
observers	*Parties*	Observers.

Choice Archive
Controller: providers
Returns: ()
(no fields)
interface instance *I* **for** *LedgerTime*

Data Types

type *T* = *LedgerTime*
Type synonym for `LedgerTime`.

1.22.3.22 Daml.Finance.Holding.Fungible

Templates

template *Factory*

Implementation of a factory template for fungible holdings.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *F* **for** *Factory*
interface instance *I* **for** *Factory*

template *Fungible*

Implementation of a fungible holding. The `Fungible` template implements the interface `Fungible.I` (which requires `Transferable.I`, `Base.I` and `Disclosure.I` to be implemented).

Signatory: (DA.Internal.Record.getField @"custodian" account), (DA.Internal.Record.getField @"owner" account), getLockers this

Field	Type	Description
instrument	*InstrumentKey*	The instrument of which units are held.
account	*AccountKey*	The account at which the holding is held. Defines the holding's owner and custodian.
amount	Decimal	Number of units.
lock	Optional *Lock*	An optional lock of a holding.
observers	*PartiesMap*	Observers.

Choice Archive
Controller: (DA.Internal.Record.getField @"custodian" account), (DA.Internal.Record.getField @"owner" account), getLockers this
Returns: ()
(no fields)
interface instance *I* **for** *Fungible*
interface instance *I* **for** *Fungible*
interface instance *I* **for** *Fungible*
interface instance *I* **for** *Fungible*

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

type *T* = *Fungible*
Type synonym for `Fungible`.

1.22.3.23 Daml.Finance.Holding.NonFungible

Templates

template *Factory*

Implementation of a factory template for non-fungible holdings.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *F* **for** *Factory*
interface instance *I* **for** *Factory*

template *NonFungible*

Implementation of a non-fungible holding. `NonFungible` implements the interface `Transferable.I` (which requires `Base.I` and `Disclosure.I` to be implemented).

Signatory: (DA.Internal.Record.getField @"custodian" account), (DA.Internal.Record.getField @"owner" account), getLockers this

Field	Type	Description
instrument	*InstrumentKey*	The instrument of which units are held.
account	*AccountKey*	The account at which the holding is held. Defines the holding's owner and custodian.
amount	Decimal	Number of units.
lock	Optional *Lock*	An optional lock of a holding.
observers	*PartiesMap*	Observers.

Choice Archive
Controller: (DA.Internal.Record.getField @"custodian" account), (DA.Internal.Record.getField @"owner" account), getLockers this
Returns: ()
(no fields)
interface instance *I* **for** *NonFungible*
interface instance *I* **for** *NonFungible*
interface instance *I* **for** *NonFungible*

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

type *T* = *NonFungible*
Type synonym for `NonFungible`.

1.22.3.24 Daml.Finance.Holding.NonTransferable

Templates

template *Factory*

Implementation of a factory template for non-transferable holdings.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *F* **for** *Factory*
interface instance *I* **for** *Factory*

template *NonTransferable*

Implementation of a non-transferable holding. `NonTransferable` implements the interface `Base.I` (which requires `Disclosure.I` to be implemented).

Signatory: (DA.Internal.Record.getField @"custodian" account), (DA.Internal.Record.getField @"owner" account), getLockers this

Field	Type	Description
instrument	*InstrumentKey*	The instrument of which units are held.
account	*AccountKey*	The account at which the holding is held. Defines the holding's owner and custodian.
amount	Decimal	Number of units.
lock	Optional *Lock*	An optional lock of a holding.
observers	*PartiesMap*	Observers.

Choice Archive
Controller: (DA.Internal.Record.getField @"custodian" account), (DA.Internal.Record.getField @"owner" account), getLockers this
Returns: ()
(no fields)
interface instance *I* **for** *NonTransferable*
interface instance *I* **for** *NonTransferable*

Data Types

type ***F*** = *Factory*
Type synonym for `Factory`.

type ***T*** = *NonTransferable*
Type synonym for `NonTransferable`.

1.22.3.25 Daml.Finance.Holding.Util

Functions

transferImpl : *I* -> ContractId *I* -> *Transfer* -> Update (ContractId *I*)
Default implementation of `transfer` for the `Transferable` interface.

acquireImpl : (HasCreate t, HasSignatory t, HasFromInterface t *I*, HasToInterface t *I*) => Optional *Lock* -> (Optional *Lock* -> t) -> *Acquire* -> Update (ContractId *I*)
Default implementation of `acquire` from the `Base` interface.

releaseImpl : (HasCreate t, HasFromInterface t *I*, HasToInterface t *I*) => Optional *Lock* -> (Optional *Lock* -> t) -> *Release* -> Update (ContractId *I*)
Default implementation of `release` from the `Base` interface.

splitImpl : (HasCreate t, HasToInterface t *I*) => Decimal -> (Decimal -> t) -> *Split* -> Update *SplitResult*
Default implementation of `split` from the `Fungible` interface.

mergeImpl : (HasCreate t, HasArchive t, HasSignatory t, HasFromInterface t *I*, HasToInterface t *I*) => Decimal -> (t -> Decimal) -> (Decimal -> t) -> *Merge* -> Update (ContractId *I*)
Default implementation of `merge` from the `Fungible` interface.

1.22.3.26 Daml.Finance.Instrument.Bond.Callable.Factory

Templates

template *Factory*

Factory template for instrument creation.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *Factory* **for** *Factory*
interface instance *I* **for** *Factory*

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

1.22.3.27 Daml.Finance.Instrument.Bond.Callable.Instrument

Templates

template *Instrument*

This template models a callable bond. It pays a fixed or a floating coupon rate at the end of every coupon period (unless the bond has been called). Callability is restricted to some (or all) of the coupon dates. This is specified by a separate schedule.

Signatory: depository, issuer

Field	Type	Description
depository	Party	The depository of the instrument.
issuer	Party	The issuer of the instrument.
id	*Id*	The identifier of the instrument.
version	Text	The instrument's version.
description	Text	A description of the instrument.
floatingRate	Optional *FloatingRate*	A description of the floating rate to be used (if applicable). This supports both Libor and SOFR style reference rates.
couponRate	Decimal	The fixed coupon rate, per annum. For example, in case of a "3.5% p.a coupon" this should be 0.035. This can also used as a floating coupon spread. For example, in case of "3M Libor + 0.5%" this should be 0.005.
capRate	Optional Decimal	The maximum coupon rate possible. For example, if "3M Libor + 0.5%" would result in a rate of 2.5%, but capRate is 2.0%, the coupon rate used would be 2.0%.
floorRate	Optional Decimal	The minimum coupon rate possible. For example, if "3M Libor + 0.5%" would result in a rate of -0.2%, but floorRate is 0.0%, the coupon rate used would be 0.0%.
couponSchedule	*PeriodicSchedule*	The schedule for the periodic coupon payments. The coupon is paid on the *last* date of each schedule period. In case of a floating rate, the reference rate will be fixed in relation to this schedule (in case of a Libor rate: at the start/end of each period, as specified by FloatingRate). This is the main schedule of the instrument, which drives both the calculation and the payment of coupons. It also defines the issue date and the maturity date of the bond.
callSchedule	*PeriodicSchedule*	The bond is callable on the *last* date of each schedule period. For example, if this schedule is the same as the couponSchedule, it means that the bond can be called on each coupon payment date.
noticeDays	Int	The number of business days in advance of the coupon date that the issuer must give notice if it wants to call the bond. The election whether to call or not to call must be done by this date.
holidayCalendarIds	[Text]	The identifiers of the holiday calendars to be used for the coupon schedule.
calendarDataProvider	Party	The reference data provider to use for the holiday calendar.
dayCountConvention	*DayCountConventionEnum*	The day count convention used to calculate day count fractions. For example: Act360.
useAdjustedDatesForDcf	Bool	Configure whether to use adjusted dates (as specified in *businessDayAdjustment* of the *couponSchedule*) for day count fractions.
currency	*InstrumentKey*	The currency of the bond. For example, if the bond pays in USD this should be a USD cash instrument.
notional	Decimal	The notional of the bond. This is the face value

Choice Archive
Controller: depository, issuer
Returns: ()
(no fields)
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*

Data Types

type *T* = *Instrument*
Type synonym for `Instrument`.

1.22.3.28 Daml.Finance.Instrument.Bond.FixedRate.Factory

Templates

template *Factory*

Factory template for instrument creation.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *Factory* **for** *Factory*
interface instance *I* **for** *Factory*

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

1.22.3.29 Daml.Finance.Instrument.Bond.FixedRate.Instrument

Templates

template *Instrument*

This template models a fixed rate bond. It pays a fixed coupon rate at the end of every coupon period.

Signatory: depository, issuer

Field	Type	Description
depository	Party	The depository of the instrument.
issuer	Party	The issuer of the instrument.
id	*Id*	The identifier of the instrument.
version	Text	The instrument's version.
description	Text	A description of the instrument.
couponRate	Decimal	The fixed coupon rate, per annum. For example, in case of a "3.5% p.a coupon" this should be 0.035.
periodicSchedule	*PeriodicSchedule*	The schedule for the periodic coupon payments.
holidayCalendarIds	[Text]	The identifiers of the holiday calendars to be used for the coupon schedule.
calendarDataProvider	Party	The reference data provider to use for the holiday calendar.
dayCountConvention	*DayCountConventionEnum*	The day count convention used to calculate day count fractions. For example: Act360.
currency	*InstrumentKey*	The currency of the bond. For example, if the bond pays in USD this should be a USD cash instrument.
notional	Decimal	The notional of the bond. This is the face value corresponding to one unit of the bond instrument. For example, if one bond unit corresponds to 1000 USD, this should be 1000.0.
observers	*PartiesMap*	The observers of the instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used. FIXED_RATE_BOND_TEMPLATE_END

Choice Archive
Controller: depository, issuer
Returns: ()
(no fields)
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*

Data Types

type *T* = *Instrument*
Type synonym for `Instrument`.

1.22.3.30 Daml.Finance.Instrument.Bond.FloatingRate.Factory

Templates

template *Factory*

Factory template for instrument creation.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *Factory* **for** *Factory*
interface instance *I* **for** *Factory*

1.22.3.31 Daml.Finance.Instrument.Bond.FloatingRate.Instrument

Templates

template *Instrument*

This template models a floating rate bond. It pays a floating coupon rate at the end of every coupon period. This consists of a reference rate (observed at the beginning of the coupon period) plus a coupon spread. For example, 3M Euribor + 0.5%.

Signatory: depository, issuer

Field	Type	Description
depository	Party	The depository of the instrument.
issuer	Party	The issuer of the instrument.
id	*Id*	An identifier of the instrument.
version	Text	The instrument's version.
description	Text	A description of the instrument.
referenceRateId	Text	The floating rate reference ID. For example, in case of "3M Euribor + 0.5%" this should be a valid reference to the "3M Euribor" reference rate. FLOATING_RATE_BOND_TEMPLATE_UNTIL_REFRATE_END
couponSpread	Decimal	The floating rate coupon spread. For example, in case of "3M Euribor + 0.5%" this should be 0.005.
periodicSchedule	*PeriodicSchedule*	The schedule for the periodic coupon payments.
holidayCalendarIds	[Text]	The identifiers of the holiday calendars to be used for the coupon schedule.
calendarDataProvider	Party	The reference data provider to use for the holiday calendar.
dayCountConvention	*DayCountConventionEnum*	The day count convention used to calculate day count fractions. For example: Act360.
currency	*InstrumentKey*	The currency of the bond. For example, if the bond pays in USD this should be a USD cash instrument.
notional	Decimal	The notional of the bond. This is the face value corresponding to one unit of the bond instrument. For example, if one bond unit corresponds to 1000 USD, this should be 1000.0.
observers	*PartiesMap*	The observers of the instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

Choice Archive
Controller: depository, issuer
Returns: ()
(no fields)
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*

Data Types

type *T* = *Instrument*
Type synonym for `Instrument`.

1.22.3.32 Daml.Finance.Instrument.Bond.InflationLinked.Factory

Templates

template *Factory*

Factory template for instrument creation.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *Factory* **for** *Factory*
interface instance *I* **for** *Factory*

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

1.22.3.33 Daml.Finance.Instrument.Bond.InflationLinked.Instrument

Templates

template *Instrument*

This template models an inflation linked bond. It pays an inflation adjusted coupon at the end of every coupon period. The coupon is based on a fixed rate, which is applied to a principal that is adjusted according to an inflation index, for example the Consumer Price Index (CPI) in the U.S. For example: 0.5% p.a. coupon, CPI adjusted principal: At maturity, the greater of the adjusted principal and the original principal is redeemed. For clarity, this only applies to the redemption amount. The coupons are always calculated based on the adjusted principal.

Signatory: depository, issuer

Field	Type	Description
depository	Party	The depository of the instrument.
issuer	Party	The issuer of the instrument.
id	*Id*	An identifier of the instrument.
version	Text	The instrument's version.
description	Text	A description of the instrument.
inflationIndexId	Text	The inflation index reference ID. For example, in case of "0.5% p.a. coupon, CPI adjusted principal" this should be a valid reference to the "CPI" index.
inflationIndexBaseValue	Decimal	The value of the inflation index on the first reference date of this bond (called "dated date" on US TIPS). This is used as the base value for the principal adjustment.
couponRate	Decimal	The fixed coupon rate, per annum. For example, in case of a "0.5% p.a. coupon, CPI adjusted principal" this should be 0.005.
periodicSchedule	*PeriodicSchedule*	The schedule for the periodic coupon payments.
holidayCalendarIds	[Text]	The identifiers of the holiday calendars to be used for the coupon schedule.
calendarDataProvider	Party	The reference data provider to use for the holiday calendar.
dayCountConvention	*DayCountConventionEnum*	The day count convention used to calculate day count fractions. For example: Act360.
currency	*InstrumentKey*	The currency of the bond. For example, if the bond pays in USD this should be a USD cash instrument.
notional	Decimal	The notional of the bond. This is the face value corresponding to one unit of the bond instrument. For example, if one bond unit corresponds to 1000 USD, this should be 1000.0.
observers	*PartiesMap*	The observers of the instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

Choice Archive
Controller: depository, issuer
Returns: ()
(no fields)
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*

Data Types

type *T* = *Instrument*
Type synonym for `Instrument`.

1.22.3.34 Daml.Finance.Instrument.Bond.Util

Data Types

type *C* = *Claim* Date Decimal *Deliverable Observable*

type *O* = *Observation* Date Decimal *Observable*

Functions

includes : *Schedule* -> *Schedule* -> [Bool]
Find out which schedule periods of scheduleA exist in scheduleB.

createCallableBondClaims : (Date -> Time) -> *Schedule* -> *Schedule* -> *PeriodicSchedule* -> Bool -> Decimal -> *DayCountConventionEnum* -> Decimal -> *Deliverable* -> Optional *FloatingRate* -> Optional Decimal -> Optional Decimal -> Int -> *HolidayCalendarData* -> *TaggedClaim*
Calculate the claims for a callable bond with a fixed and/or floating coupon on each payment date and a redemption amount at the end (unless called by the issuer).

1.22.3.35 Daml.Finance.Instrument.Bond.ZeroCoupon.Factory

Templates

template *Factory*

Factory template for instrument creation.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *Factory* **for** *Factory*
interface instance *I* **for** *Factory*

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

1.22.3.36 Daml.Finance.Instrument.Bond.ZeroCoupon.Instrument

Templates

template *Instrument*

This template models a zero coupon bond. It does not pay any coupons, only the redemption amount at maturity.

Signatory: depository, issuer

Field	Type	Description
depository	Party	The depository of the instrument.
issuer	Party	The issuer of the instrument.
id	*Id*	An identifier of the instrument.
version	Text	The instrument's version.
description	Text	A description of the instrument.
issueDate	Date	The date when the bond was issued.
maturityDate	Date	The redemption date of the bond.
currency	*InstrumentKey*	The currency of the bond. For example, if the bond pays in USD this should be a USD cash instrument.
notional	Decimal	The notional of the bond. This is the face value corresponding to one unit of the bond instrument. For example, if one bond unit corresponds to 1000 USD, this should be 1000.0.
observers	*PartiesMap*	The observers of the instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

Choice Archive
Controller: depository, issuer
Returns: ()
(no fields)
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*

Data Types

type *T* = *Instrument*
Type synonym for `Instrument`.

1.22.3.37 Daml.Finance.Instrument.Equity.Factory

Templates

template *Factory*

Factory template for instrument creation.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *F* **for** *Factory*
interface instance *I* **for** *Factory*

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

1.22.3.38 Daml.Finance.Instrument.Equity.Instrument

Templates

template *Instrument*

An Instrument representing a common stock.

Signatory: depository, issuer

Field	Type	Description
issuer	Party	Issuer.
depository	Party	Depository.
id	*Id*	A textual identifier.
version	Text	The instrument's version.
description	Text	A description of the instrument.
observers	*PartiesMap*	Observers.
validAsOf	Time	Timestamp as of which the instrument is valid. This usually coincides with the timestamp of the event that creates the instrument. It usually does not coincide with ledger time.

Choice Archive
Controller: depository, issuer
Returns: ()
(no fields)
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*

Data Types

type *T* = *Instrument*
Type synonym for `Instrument`.

1.22.3.39 Daml.Finance.Instrument.Generic.Factory

Templates

template *Factory*

Factory template for generic instrument creation.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *F* **for** *Factory*
interface instance *I* **for** *Factory*

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

1.22.3.40 Daml.Finance.Instrument.Generic.Instrument

Templates

template *Instrument*

An instrument representing a generic payoff, modelled using the Contingent Claims library. The responsibility for processing lifecycle events as well as elections is delegated to the issuer, who is hence responsible for providing the correct `Observable`\s.

Signatory: depository, issuer

Field	Type	Description
depository	Party	The instrument depository.
issuer	Party	The instrument issuer.
id	*Id*	The identifier with corresponding version.
version	Text	The instrument's version.
description	Text	A human readable description of the instrument.
claims	*C*	The claim tree.
acquisitionTime	Time	The claim's acquisition time. This usually corresponds to the start date of the contract.
observers	*PartiesMap*	Observers.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

Choice Archive
Controller: depository, issuer
Returns: ()
(no fields)
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*

Data Types

type *T* = *Instrument*
Type synonym for `Instrument`.

1.22.3.41 Daml.Finance.Instrument.Generic.Lifecycle.Rule

Templates

template *Rule*

Rule to process a lifecycle event. This rule supports both time update events and election events.

Signatory: providers

Field	Type	Description
providers	*Parties*	Providers of the distribution rule.
lifecycler	Party	Party performing the lifecycling.
observers	*PartiesMap*	Observers of the distribution rule.
id	*Id*	Identifier for the rule contract.
description	Text	Textual description.

Choice Archive
Controller: providers
Returns: ()
(no fields)

1.22.3.42 Daml.Finance.Instrument.Option.BarrierEuropeanCash.Factory

Templates

template *Factory*

Factory template for instrument creation.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *Factory* **for** *Factory*
interface instance *I* **for** *Factory*

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

1.22.3.43 Daml.Finance.Instrument.Option.BarrierEuropeanCash.Instrument

Templates

template *Instrument*

This template models a cash settled, automatically excercised barrier option with European exercise.

Signatory: depository, issuer

Field	Type	Description
depository	Party	The depository of the instrument.
issuer	Party	The issuer of the instrument.
id	*Id*	An identifier of the instrument.
version	Text	The instrument's version.
description	Text	A description of the instrument.
referenceAssetId	Text	The reference asset ID. For example, in case of an option on AAPL this should be a valid reference to the AAPL fixings to be used for the payoff calculation.
ownerReceives	Bool	Indicate whether a holding owner of this instrument receives option payoff.
optionType	*OptionTypeEnum*	Indicate whether the option is a call or a put.
strike	Decimal	The strike price of the option.
barrier	Decimal	The barrier price of the option.
barrierType	*BarrierTypeEnum*	The type of barrier.
barrierStartDate	Date	The start date for barrier observations.
expiryDate	Date	The expiry date of the option.
currency	*InstrumentKey*	The currency of the option. For example, if the option pays in USD this should be a USD cash instrument.
observers	*PartiesMap*	The observers of the instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

Choice Archive
Controller: depository, issuer
Returns: ()
(no fields)
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*

interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*

Data Types

type *T* = *Instrument*
Type synonym for `Instrument`.

1.22.3.44 Daml.Finance.Instrument.Option.Dividend.Factory

Templates

template *Factory*

Factory template for instrument creation.

Signatory: provider

Field	Type	Description
provider	*Party*	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *Factory* **for** *Factory*
interface instance *I* **for** *Factory*

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

1.22.3.45 Daml.Finance.Instrument.Option.Dividend.Instrument

Templates

template *Instrument*

This template models a physically settled Dividend option. The holder gets to choose to receive the dividend in cash or in a different form (in shares and/or in a foreign currency).

Signatory: depository, issuer

Field	Type	Description
depository	Party	The depository of the instrument.
issuer	Party	The issuer of the instrument.
id	*Id*	An identifier of the instrument.
version	Text	The instrument's version.
description	Text	A description of the instrument.
expiryDate	Date	The expiry date of the option.
cashQuantity	*InstrumentQuantity*	Dividend paid in cash
sharesQuantity	Optional *InstrumentQuantity*	Dividend paid in shares (if applicable)
fxQuantity	Optional *InstrumentQuantity*	Dividend paid in a foreign currency (if applicable)
observers	*PartiesMap*	The observers of the instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.
prevElections	[EventData]	A list of previous elections that have been lifecycled on this instrument so far.

Choice Archive
Controller: depository, issuer
Returns: ()
(no fields)
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*

Data Types

type *T* = *Instrument*
Type synonym for `Instrument`.

1.22.3.46 Daml.Finance.Instrument.Option.Dividend.Election

Templates

template *Factory*

Factory template to create an `Election`.

Signatory: provider

Field	Type	Description
provider	Party	The provider of the `Factory`.
observers	*PartiesMap*	A set of observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *F* **for** *Factory*
interface instance *I* **for** *Factory*

1.22.3.47 Daml.Finance.Instrument.Option.EuropeanCash.Factory

Templates

template *Factory*

Factory template for instrument creation.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *Factory* **for** *Factory*
interface instance *I* **for** *Factory*

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

1.22.3.48 Daml.Finance.Instrument.Option.EuropeanCash.Instrument

Templates

template *Instrument*

This template models a cash settled, automatically excercised European option.

Signatory: depository, issuer

Field	Type	Description
depository	Party	The depository of the instrument.
issuer	Party	The issuer of the instrument.
id	*Id*	An identifier of the instrument.
version	Text	The instrument's version.
description	Text	A description of the instrument.
referenceAssetId	Text	The reference asset ID. For example, in case of an option on AAPL this should be a valid reference to the AAPL fixings to be used for the payoff calculation.
ownerReceives	Bool	Indicate whether a holding owner of this instrument receives option payoff.
optionType	*OptionTypeEnum*	Indicate whether the option is a call or a put.
strike	Decimal	The strike price of the option.
expiryDate	Date	The expiry date of the option.
currency	*InstrumentKey*	The currency of the option. For example, if the option pays in USD this should be a USD cash instrument.
observers	*PartiesMap*	The observers of the instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

Choice Archive
Controller: depository, issuer
Returns: ()
(no fields)
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*

Data Types

type *T* = *Instrument*
Type synonym for `Instrument`.

1.22.3.49 Daml.Finance.Instrument.Option.EuropeanPhysical.Factory

Templates

template *Factory*

Factory template for instrument creation.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *Factory* **for** *Factory*
interface instance *I* **for** *Factory*

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

1.22.3.50 Daml.Finance.Instrument.Option.EuropeanPhysical.Instrument

Templates

template *Instrument*

This template models a physically settled European option.

Signatory: depository, issuer

Field	Type	Description
depository	Party	The depository of the instrument.
issuer	Party	The issuer of the instrument.
id	*Id*	An identifier of the instrument.
version	Text	The instrument's version.
description	Text	A description of the instrument.
referenceAsset	*InstrumentKey*	The reference asset. For example, in case of an option on AAPL this should be an AAPL instrument.
ownerReceives	Bool	Indicate whether a holding owner of this instrument receives option payoff.
optionType	*OptionType-Enum*	Indicate whether the option is a call or a put.
strike	Decimal	The strike price of the option.
expiryDate	Date	The expiry date of the option.
currency	*InstrumentKey*	The currency of the option. For example, if the option pays in USD this should be a USD cash instrument.
observers	*PartiesMap*	The observers of the instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.
prevElections	[EventData]	A list of previous elections that have been lifecycled on this instrument so far.

Choice Archive
Controller: depository, issuer
Returns: ()
(no fields)
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*

Data Types

type *T* = *Instrument*
Type synonym for `Instrument`.

1.22.3.51 Daml.Finance.Instrument.Option.Util

Functions

dateToDateClockTime : Date -> Time
Maps a `Date` to `Time` using the rule in the `DateClock`.

1.22.3.52 Daml.Finance.Instrument.Swap.Asset.Factory

Templates

template *Factory*

Factory template for instrument creation.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *Factory* **for** *Factory*
interface instance *I* **for** *Factory*

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

1.22.3.53 Daml.Finance.Instrument.Swap.Asset.Instrument

Templates

template *Instrument*

This template models an asset swap. It pays an asset performance vs a fix interest rate at the end of every payment period. It can be used to model equity swaps, some types of commodity swaps (of the form performance vs rate) and swaps with the same payoff on other asset types. The asset leg is described by an observable containing either unadjusted or adjusted fixings (for a price return or a total return swap, respectively). The template calculates the performance for each payment period using this observable. For example: AAPL total return vs 2.5% fix.

Signatory: depository, issuer

Field	Type	Description
depository	Party	The depository of the instrument.
issuer	Party	The issuer of the instrument.
id	*Id*	An identifier of the instrument.
version	Text	The instrument's version.
description	Text	A description of the instrument.
referenceAssetId	Text	The reference asset ID. For example, in case of "AAPL total return vs 2.5% fix" this should be a valid reference to the AAPL fixings to be used for the total return calculation (dividend-adjusted fixings).
ownerReceivesFix	Bool	Indicate whether a holding owner of this instrument receives the fix or the asset leg of the swap.
fixRate	Decimal	The interest rate of the fix leg. For example, in case of "AAPL total return vs 2.5% fix" this should be 0.025.
periodicSchedule	*PeriodicSchedule*	The schedule for the periodic swap payments.
holidayCalendarIds	[Text]	The identifiers of the holiday calendars to be used for the swap payment schedule.
calendarDataProvider	Party	The reference data provider to use for the holiday calendar.
dayCountConvention	*DayCountConventionEnum*	The day count convention used to calculate day count fractions. For example: Act360.
currency	*InstrumentKey*	The currency of the swap. For example, if the swap pays in USD this should be a USD cash instrument.
observers	*PartiesMap*	The observers of the instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

Choice Archive
Controller: depository, issuer
Returns: ()
(no fields)
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*

Data Types

type *T* = *Instrument*
Type synonym for `Instrument`.

1.22.3.54 Daml.Finance.Instrument.Swap.CreditDefault.Factory

Templates

template *Factory*

Factory template for instrument creation.

Signatory: provider

Field	Type	Description
provider	*Party*	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *Factory* **for** *Factory*
interface instance *I* **for** *Factory*

1.22.3.55 Daml.Finance.Instrument.Swap.CreditDefault.Instrument

Templates

template *Instrument*

This template models a cash-settled credit default swap. In case of a credit default event it pays (1-recoveryRate), in exchange for a fix rate at the end of every payment period. For example: 2.5% fix vs (1-recoveryRate) if TSLA defaults on a bond payment

Signatory: depository, issuer

Field	Type	Description
depository	Party	The depository of the instrument.
issuer	Party	The issuer of the instrument.
id	*Id*	An identifier of the instrument.
version	Text	The instrument's version.
description	Text	A description of the instrument.
defaultProbabilityReferenceId	Text	The reference ID of the default probability observable. For example, in case of protection against a "TSLA bond payment default" this should be a valid reference to the "TSLA default probability".
recoveryRateReferenceId	Text	The reference ID of the recovery rate observable. For example, in case of a "TSLA bond payment default with a 60% recovery rate" this should be a valid reference to the "TSLA bond recovery rate".
ownerReceivesFix	Bool	Indicate whether a holding owner of this instrument receives the fix or the default protection leg of the swap.
fixRate	Decimal	The interest rate of the fix leg. For example, in case of "2.5% fix" this should be 0.025.
periodicSchedule	*PeriodicSchedule*	The schedule for the periodic swap payments.
holidayCalendarIds	[Text]	The identifiers of the holiday calendars to be used for the swap payment schedule.
calendarDataProvider	Party	The reference data provider to use for the holiday calendar.
dayCountConvention	*DayCountConventionEnum*	The day count convention used to calculate day count fractions. For example: Act360.
currency	*InstrumentKey*	The currency of the swap. For example, if the swap pays in USD this should be a USD cash instrument.
observers	*PartiesMap*	The observers of the instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

Choice Archive
Controller: depository, issuer
Returns: ()
(no fields)
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*

Data Types

type *T* = *Instrument*
Type synonym for `Instrument`.

1.22.3.56 Daml.Finance.Instrument.Swap.Currency.Factory

Templates

template *Factory*

Factory template for instrument creation.

Signatory: provider

Field	Type	Description
provider	*Party*	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *Factory* **for** *Factory*
interface instance *I* **for** *Factory*

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

1.22.3.57 Daml.Finance.Instrument.Swap.Currency.Instrument

Templates

template *Instrument*

This template models a currency swap. It pays a fix vs fix rate (in different currencies) at the end of every payment period. The principal in the foreign currency is calculated using an fx rate and the principal amount in the base currency. The principal is not exchanged. For example: USD 1000k principal, fx rate 1.10 -> EUR 1100k principal 3% fix rate on USD 1000k vs 2% fix rate on EUR 1100k

Signatory: depository, issuer

Field	Type	Description
depository	Party	The depository of the instrument.
issuer	Party	The issuer of the instrument.
id	*Id*	An identifier of the instrument.
version	Text	The instrument's version.
description	Text	A description of the instrument.
ownerReceivesBase	Bool	Indicate whether a holding owner of this instrument receives the base currency leg or the foreign currency leg of the swap.
baseRate	Decimal	The interest rate of the base currency. For example, in case of "3% in USD" this should be 0.03.
foreignRate	Decimal	The interest rate of the foreign currency. For example, in case of "2% in EUR" this should be 0.02.
periodicSchedule	*PeriodicSchedule*	The schedule for the periodic swap payments.
holidayCalendarIds	[Text]	The identifiers of the holiday calendars to be used for the swap payment schedule.
calendarDataProvider	Party	The reference data provider to use for the holiday calendar.
dayCountConvention	*DayCountConventionEnum*	The day count convention used to calculate day count fractions. For example: Act360.
baseCurrency	*InstrumentKey*	The base currency of the swap. For example, in the case of USD this should be a USD cash instrument.
foreignCurrency	*InstrumentKey*	The foreign currency of the swap. For example, in case of EUR this should be a EUR cash instrument.
fxRate	Decimal	The fx rate used to convert from the base currency principal amount to the foreign currency principal amount.
observers	*PartiesMap*	The observers of the instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

Choice Archive
Controller: depository, issuer
Returns: ()
(no fields)
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*

Data Types

type *T* = *Instrument*
Type synonym for `Instrument`.

1.22.3.58 Daml.Finance.Instrument.Swap.ForeignExchange.Factory

Templates

template *Factory*

Factory template for instrument creation.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *Factory* **for** *Factory*
interface instance *I* **for** *Factory*

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

1.22.3.59 Daml.Finance.Instrument.Swap.ForeignExchange.Instrument

Templates

template *Instrument*

This template models a foreign exchange swap (FX Swap). It has two legs: an initial FX transaction and a final FX transaction. The instrument has a base currency and a foreign currency. The convention is that a holding owner receives the foreign currency in the initial transaction (and pays it in the final transaction). Both FX rates and transaction dates are predetermined between the counterparties. For example: USD 1000k vs EUR 1100k (fx rate: 1.10) today USD 1000k vs EUR 1200k (fx rate: 1.20) in 6 months

Signatory: depository, issuer

Field	Type	Description
depository	Party	The depository of the instrument.
issuer	Party	The issuer of the instrument.
id	*Id*	An identifier of the instrument.
version	Text	The instrument's version.
description	Text	A description of the instrument.
firstFxRate	Decimal	The fx rate used for the first swap payment.
finalFxRate	Decimal	The fx rate used for the final swap payment.
issueDate	Date	The date when the swap was issued.
firstPaymentDate	Date	The first payment date of the swap.
maturityDate	Date	The final payment date of the swap.
baseCurrency	*InstrumentKey*	The base currency of the swap, which will be exchanged to another (foreign) currency on the first payment date. For example, in case of USD this should be a USD cash instrument.
foreignCurrency	*InstrumentKey*	The foreign currency of the swap. For example, in case of EUR this should be a EUR cash instrument.
observers	*PartiesMap*	The observers of the instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

Choice Archive
Controller: depository, issuer
Returns: ()
(no fields)
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*

Data Types

type *T* = *Instrument*
Type synonym for `Instrument`.

1.22.3.60 Daml.Finance.Instrument.Swap.Fpml.Factory

Templates

template *Factory*

Factory template for instrument creation.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *Factory* **for** *Factory*
interface instance *I* **for** *Factory*

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

1.22.3.61 Daml.Finance.Instrument.Swap.Fpml.Instrument

Templates

template *Instrument*

This template models a swap specified by FpML swapStream modules. It can contain one or several legs of different types: fix or floating rates

Signatory: depository, issuer

Field	Type	Description
depository	Party	The depository of the instrument.
issuer	Party	The issuer of the instrument.
id	*Id*	An identifier of the instrument.
version	Text	The instrument's version.
description	Text	A description of the instrument.
swapStreams	[*SwapStream*]	Each element describes a stream of swap payments, for example a regular fixed or floating rate.
issuerPartyRef	Text	Used to the identify which counterparty is the issuer in the swapStream.
calendarDataProvider	Party	The reference data provider to use for the holiday calendar.
currencies	[*InstrumentKey*]	The currencies of the different swap legs, one for each swapStream. For example, if one leg pays in USD this should be a USD cash instrument.
observers	*PartiesMap*	The observers of the instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

Choice Archive
Controller: depository, issuer
Returns: ()
(no fields)
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*

Data Types

type *T* = *Instrument*
Type synonym for `Instrument`.

1.22.3.62 Daml.Finance.Instrument.Swap.Fpml.Util

Data Types

type *C* = *Claim* Date Decimal *Deliverable Observable*

type *O* = *Observation* Date Decimal *Observable*

Functions

createCalculationPeriodicSchedule : *CalculationPeriodDates* -> *PeriodicSchedule*
Create a schedule for calculation periods.

createPaymentPeriodicSchedule : *SwapStream* -> *PeriodicSchedule*
Create a schedule for payment periods.

getCalendarsAndAdjust : Date -> *BusinessDayAdjustments* -> Party -> Party -> Update Date
Retrieve holiday calendars and adjust a date as specified in a BusinessDayAdjustments FpML element

adjustDateAccordingToBusinessDayAdjustments : Date -> *BusinessDayAdjustments* -> Party -> Party -> Update Date
Adjust a date as specified in a BusinessDayAdjustments FpML element (or not at all if NoAdjustment)

applyPaymentDaysOffset : [*SchedulePeriod*] -> *PaymentDates* -> [*HolidayCalendarData*] -> [*SchedulePeriod*]
Adjust payment schedule according to paymentDaysOffset (if available).

getSingleStubRate : *StubFloatingRate* -> Date -> Optional *O*
Define observable part of claim when one specific floating rate is provided for a stub period.

getInterpolatedStubRate : *StubFloatingRate* -> *StubFloatingRate* -> *SchedulePeriod* -> *HolidayCalendarData* -> *BusinessDayConventionEnum* -> Date -> Optional *O*
Linearly interpolates two rates within a period, as specified in https://www.isda.org/a/aWkgE/Linear-interpolation-04022022.pdf

getStubRateFloating : [*StubFloatingRate*] -> *SchedulePeriod* -> *HolidayCalendarData* -> *BusinessDayConventionEnum* -> Date -> Optional *O*
Get the floating stub rate to be used for a stub period.

getStubRate : *StubCalculationPeriodAmount* -> Bool -> *SchedulePeriod* -> *HolidayCalendarData* -> *BusinessDayConventionEnum* -> Bool -> Date -> Optional *O*
Get the stub rate to be used for a stub period. Currently, three main options from the FpML schema are supported:

1. A fix stubRate.
2. One or two floating rates for the stub.
3. No specific stub rate defined -> use the same rate as is used for regular periods.

alignPaymentSchedule : [*SchedulePeriod*] -> [*SchedulePeriod*] -> Update [*SchedulePeriod*]
Align the payment schedule with the calculation schedule.

verifyFxScheduleAndGetId : [*SchedulePeriod*] -> *SwapStream* -> Party -> Party -> *FxLinkedNotionalSchedule* -> Update (Optional Text, Optional Decimal, Optional [Date])

getFxRateId : [*SchedulePeriod*] -> *SwapStream* -> Party -> Party -> Update (Optional Text, Optional Decimal, Optional [Date])

getRateFixingsAndCalendars : *SwapStream* -> *ResetDates* -> [*SchedulePeriod*] -> Party -> Party -> Update ([Date], *HolidayCalendarData*)

calculateFixPaymentClaimsFromSwapStream : *FixedRateSchedule* -> *SwapStream* -> *PeriodicSchedule* -> [*SchedulePeriod*] -> [*SchedulePeriod*] -> Bool -> Bool -> *Deliverable* -> Party -> Party -> Optional Text -> Optional [Date] -> [(Decimal, Bool)] -> Update [*TaggedClaim*]
Create claims from swapStream that describes a fixed coupon stream.

calculatePrincipalExchangePaymentClaims : [*SchedulePeriod*] -> Bool -> *Deliverable* -> Optional Text -> [(Decimal, Bool)] -> [Date] -> *PrincipalExchanges* -> *TaggedClaim*
Create principal exchange claims.

roundRate : Decimal -> *Rounding* -> Decimal
Apply rounding convention to the rate used in a calculation period. Takes a Rounding FpML object as an input: https://www.fpml.org/spec/fpml-5-11-3-lcwd-1/html/confirmation/schemaDocumentation/schemas/fpr

checkRefRateCompounding : *FloatingRateCalculation* -> (Bool, Optional *DayCountConventionEnum*)
Check whether a FloatingRateCalculation uses a reference rate that needs to be compounded. Seems there is no FpML element that specificies this, but that it is implicit in the rate name, for example "USD-SOFR-COMPOUND" If it is a compounded reference rate, also return the daycount convention that was used for the corresponding reference index, e.g. Act360 in the case of the SOFR Index.

calculateFloatingPaymentClaimsFromSwapStream : *FloatingRateCalculation* -> *SwapStream* -> *PeriodicSchedule* -> [*SchedulePeriod*] -> [*SchedulePeriod*] -> Bool -> Bool -> *Deliverable* -> Party -> Party -> Optional Text -> Optional [Date] -> [(Decimal, Bool)] -> Update [*TaggedClaim*]
Create claims from swapStream that describes a floating coupon stream.

calculateClaimsFromSwapStream : *SwapStream* -> *PeriodicSchedule* -> [*SchedulePeriod*] -> [*SchedulePeriod*] -> Optional *SwapStream* -> Bool -> Bool -> *Deliverable* -> Party -> Party -> Update [*TaggedClaim*]
Create claims from swapStream that describes a fixed or floating coupon stream.

1.22.3.63 Daml.Finance.Instrument.Swap.InterestRate.Factory

Templates

template *Factory*

Factory template for instrument creation.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *Factory* **for** *Factory*
interface instance *I* **for** *Factory*

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

1.22.3.64 Daml.Finance.Instrument.Swap.InterestRate.Instrument

Templates

template *Instrument*

This template models an interest rate swap. It pays a fix vs floating rate at the end of every payment period. The floating leg depends on a reference rate (observed at the beginning of the swap payment period). For example: 3M Euribor vs 2.5% fix.

Signatory: depository, issuer

Field	Type	Description
depository	Party	The depository of the instrument.
issuer	Party	The issuer of the instrument.
id	*Id*	An identifier of the instrument.
version	Text	The instrument's version.
description	Text	A description of the instrument.
referenceRateId	Text	The floating rate reference ID. For example, in case of "3M Euribor vs 2.5% fix" this should be a valid reference to the "3M Euribor" reference rate.
ownerReceivesFix	Bool	Indicate whether a holding owner of this instrument receives the fix or the floating leg of the swap.
fixRate	Decimal	The interest rate of the fix leg. For example, in case of "3M Euribor vs 2.5% fix" this should be 0.025.
periodicSchedule	*PeriodicSchedule*	The schedule for the periodic swap payments.
holidayCalendarIds	[Text]	The identifiers of the holiday calendars to be used for the swap payment schedule.
calendarDataProvider	Party	The reference data provider to use for the holiday calendar.
dayCountConvention	*DayCountConventionEnum*	The day count convention used to calculate day count fractions. For example: Act360.
currency	*InstrumentKey*	The currency of the swap. For example, if the swap pays in USD this should be a USD cash instrument.
observers	*PartiesMap*	The observers of the instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

Choice Archive
Controller: depository, issuer
Returns: ()
(no fields)
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*

Data Types

type *T* = *Instrument*
Type synonym for `Instrument`.

1.22.3.65 Daml.Finance.Instrument.Token.Factory

Templates

template *Factory*

Factory template for instrument creation.

Signatory: provider

Field	Type	Description
provider	Party	The factory's provider.
observers	*PartiesMap*	The factory's observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *Factory* **for** *Factory*
interface instance *I* **for** *Factory*

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

1.22.3.66 Daml.Finance.Instrument.Token.Instrument

Templates

template *Instrument*

Implementation of a Token Instrument, which is a simple instrument whose economic terms on the ledger are represented by an `id` and a textual `description`.

Signatory: depository, issuer

Field	Type	Description
depository	Party	The instrument's depository.
issuer	Party	The instrument's issuer.
id	*Id*	The intrument's identifier.
version	Text	A textual instrument version.
description	Text	A description of the instrument.
validAsOf	Time	Timestamp as of which the instrument is valid. This usually coincides with the timestamp of the event that creates the instrument. It usually does not coincide with ledger time.
observers	*PartiesMap*	Observers.

Choice Archive
Controller: depository, issuer
Returns: ()
(no fields)
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*
interface instance *I* **for** *Instrument*

Data Types

type *T* = *Instrument*
Type synonym for `Instrument`.

1.22.3.67 Daml.Finance.Interface.Account.Account

We recommend to import this module qualified.

Interfaces

interface *Account*

An interface which represents an established relationship between a provider and an owner.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Credit*
Creates a new `Holding` in the corresponding `Account`.
Controller: (DA.Internal.Record.getField @"custodian" (view this)), (DA.Internal.Record.getField @"incoming" (DA.Internal.Record.getField @"controllers" (view this)))
Returns: ContractId *I*

Field	Type	Description
quantity	*Quantity InstrumentKey* Decimal	The target `Instrument` and corresponding amount.

Choice *Debit*
Removes an existing `Holding`.
Controller: (DA.Internal.Record.getField @"custodian" (view this)), (DA.Internal.Record.getField @"outgoing" (DA.Internal.Record.getField @"controllers" (view this)))
Returns: ()

Field	Type	Description
holdingCid	ContractId *I*	The `Holding`'s contract id.

Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party fetching the view.

Method credit : *Credit* -> Update (ContractId *I*)
Implementation of the `Credit` choice.
Method debit : *Debit* -> Update ()
Implementation of the `Debit` choice.
Method getKey : *AccountKey*
Get the unique key of the `Account`.

Data Types

data *Controllers*

Controllers of the account (related to transfers).

Controllers

Field	Type	Description
outgoing	*Parties*	Parties instructing a transfer (outgoing).
incoming	*Parties*	Parties approving a transfer (incoming).

instance Eq *Controllers*

instance Show *Controllers*

type *I* = *Account*
Type synonym for `Account`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type *R* = Reference

Type synonym for `Reference`. This type is currently used as a work-around given the lack of interface keys.

type *V* = *View*

Type synonym for `View`.

instance HasFromAnyView *Account V*

data *View*

View for `Account`.

View

Field	Type	Description
custodian	Party	Party providing accounting services.
owner	Party	Party owning this account.
id	*Id*	Identifier for the account.
description	Text	Human readable description of the account.
holdingFactoryCid	ContractId *F*	Associated holding factory.
controllers	*Controllers*	Parties controlling transfers.

instance Eq *View*

instance Show *View*

Functions

toKey : *View* -> *AccountKey*

Convert the account's 'View' to its key.

getKey : *Account* -> *AccountKey*

credit : *Account* -> *Credit* -> Update (ContractId *I*)

debit : *Account* -> *Debit* -> Update ()

exerciseInterfaceByKey : (HasInterfaceTypeRep i, HasExercise i d r) => *AccountKey* -> Party -> d -> Update r

Exercise interface by key. This method can be used to exercise a choice on an `Account` given its `AccountKey`. Requires as input the `AccountKey`, the actor fetching the account and the choice arguments. For example:

```
exerciseInterfaceByKey @Account.I accountKey actor Account.Debit with
↪holdingCid
```

disclose : (Text, *Parties*) -> Party -> *Parties* -> *AccountKey* -> Update (ContractId *I*)

Disclose account.

undisclose : (Text, *Parties*) -> Party -> *Parties* -> *AccountKey* -> Update (Optional (ContractId *I*))

Undisclose account.

1.22.3.68 Daml.Finance.Interface.Account.Factory

Interfaces

interface *Factory*

Interface that allows implementing templates to create accounts.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new account.
Controller: (DA.Internal.Record.getField @"custodian" account), (DA.Internal.Record.getField @"owner" account)
Returns: ContractId *I*

Field	Type	Description
account	*AccountKey*	The account's key.
holdingFactoryCid	ContractId *F*	Associated holding factory for the account.
controllers	*Controllers*	Controllers of the account.
description	Text	Human readable description of the account.
observers	*PartiesMap*	The account's observers.

Choice *Remove*
Archive an account.
Controller: (DA.Internal.Record.getField @"custodian" account), (DA.Internal.Record.getField @"owner" account)
Returns: ()

Field	Type	Description
account	*AccountKey*	The account's key.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.
Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.69 Daml.Finance.Interface.Account.Util

Functions

fetchAccount : HasToInterface t *I* => t -> Update *I*
Fetch the account of a holding.

getAccount : HasToInterface t *I* => t -> *AccountKey*
Get the account key of a holding.

getCustodian : HasToInterface t *I* => t -> Party
Get the custodian of a holding.

getOwner : HasToInterface t *I* => t -> Party
Get the owner of a holding.

1.22.3.70 Daml.Finance.Interface.Claims.Claim

Interfaces

interface *Claim*

Interface implemented by templates that can be represented as Contingent Claims.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetClaims*
Retrieves the list of claims representing the instrument. This might involve fetching reference data, such as calendars, on which the actor must have visibility.
Controller: actor
Returns: [*TaggedClaim*]

Field	Type	Description
actor	Party	The party retrieving the claims.

Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Method getClaims : *GetClaims* -> Update [*TaggedClaim*]
The list of claims representing the instrument.

Data Types

type ***I*** = *Claim*
Type synonym for `Claim`.

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Claim V*

data *View*

View for `Claim`.

View

Field	Type	Description
acquisitionTime	Time	The claim's acquisition time.

instance Eq *View*

instance Show *View*

Functions

getClaims : *Claim* -> *GetClaims* -> Update [*TaggedClaim*]

getClaim : Party -> *Claim* -> Update *C*
Retrieves the single claim representing the template. An error is thrown if there are zero or more than one claims.

getAcquisitionTime : *Claim* -> Time
Retrieves the claim's acquisition time.

1.22.3.71 Daml.Finance.Interface.Claims.Dynamic.Instrument

Interfaces

interface *Instrument*

Interface implemented by instruments that create Contingent Claims trees on-the-fly (i.e., the tree is not stored on disk as part of a contract, but created and processed in-memory).

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *CreateNewVersion*
Create a new version of an instrument, using a new lastEventTimestamp and a list of previous elections (if applicable).
Controller: (DA.Internal.Record.getField @"lifecycler" (view this))
Returns: ContractId *Instrument*

Field	Type	Description
version	Text	The new version of the instrument.
lastEventTimes-tamp	Time	The new lastEventTimestamp of the instru-ment.
prevElections	[EventData]	A list of previous elections that have been lifecycled on this instrument so far.

Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Method createNewVersion : *CreateNewVersion* -> Update (ContractId *Instrument*)

Data Types

type ***I*** = *Instrument*
Type synonym for `Instrument`.

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Instrument V*

data *View*

View for `Instrument`.

View

Field	Type	Description
lifecycler	Party	Party performing the lifecycling.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.
prevElections	[EventData]	A list of previous elections that have been lifecycled on this instrument so far.

instance Eq *View*

instance Show *View*

Functions

createNewVersion : *Instrument* -> *CreateNewVersion* -> Update (ContractId *Instrument*)

1.22.3.72 Daml.Finance.Interface.Claims.Types

Data Types

type ***C*** = *Claim Time Decimal Deliverable Observable*
The specialized claim type.

type ***Deliverable*** = *InstrumentKey*
Type used to reference assets in the claim tree.

type ***Observable*** = Text
Type used to reference observables in the claim tree.

data *Pending*

Type used to record pending payments.

Pending

Field	Type	Description
t	Time	
tag	Text	
instrument	*Deliverable*	
amount	Decimal	

instance Eq *Pending*

instance Show *Pending*

data *TaggedClaim*

A claim and a textual tag.

TaggedClaim

Field	Type	Description
claim	*C*	
tag	Text	

instance Eq *TaggedClaim*

instance Show *TaggedClaim*

instance HasMethod *Claim* "getClaims" (*GetClaims* -> Update [*TaggedClaim*])

1.22.3.73 Daml.Finance.Interface.Data.Numeric.Observation

Interfaces

interface *Observation*

Interface for a time-dependent numeric `Observation`, where the values are explicitly stored on-ledger.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party fetching the view.

Data Types

type *I* = *Observation*
Type synonym for `Observation`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Observation V*

data *View*

View for `Observation`.

View

Field	Type	Description
provider	Party	The reference data provider.
id	*Id*	A textual identifier.
observations	Map Time Decimal	The time-dependent values.
observers	*PartiesMap*	Observers.

1.22.3.74 Daml.Finance.Interface.Data.Numeric.Observation.Factory

Interfaces

interface *Factory*

Factory contract used to create, remove and view a `Numeric.Observation`.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create an `Observation`.
Controller: provider
Returns: ContractId *I*

Field	Type	Description
provider	Party	The reference data provider.
id	*Id*	A textual identifier.
observations	Map Time Decimal	The time-dependent values.
observers	*PartiesMap*	Observers.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View

Field	Type	Description
provider	Party	The provider of the `Factory`.
observers	*PartiesMap*	The observers of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

1.22.3.75 Daml.Finance.Interface.Data.Reference.HolidayCalendar

Interfaces

interface *HolidayCalendar*

Interface for contracts storing holiday calendar data on the ledger.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party fetching the view.

Choice *UpdateCalendar*
Updates the holiday calendar.
Controller: (DA.Internal.Record.getField @"provider" (view this))
Returns: ContractId *HolidayCalendar*

Field	Type	Description
newCalendar	*HolidayCalendarData*	The new `HolidayCalendarData`.

Method updateCalendar : *UpdateCalendar* -> Update (ContractId *HolidayCalendar*)
Updates the holiday calendar.

Data Types

type *I* = *HolidayCalendar*
Type synonym for `HolidayCalendar`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *HolidayCalendar V*

data *View*

View for `HolidayCalendar`.

View

Field	Type	Description
provider	Party	The parties providing the `HolidayCalendar`.
calendar	*HolidayCalendarData*	Holiday Calendar Data used to define holidays.

Functions

updateCalendar : *HolidayCalendar* -> *UpdateCalendar* -> Update (ContractId *HolidayCalendar*)

1.22.3.76 Daml.Finance.Interface.Data.Reference.HolidayCalendar.Factory

Interfaces

interface *Factory*

Interface that allows implementing templates to create holiday calendars.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new Holiday Calendar.
Controller: provider
Returns: ContractId *I*

Field	Type	Description
calendar	*HolidayCalendarData*	Holiday Calendar Data used to define holidays.
observers	*PartiesMap*	Observers.
provider	Party	The calendar's provider.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.

Data Types

type ***F*** = *Factory*
Type synonym for `Factory`.

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

1.22.3.77 Daml.Finance.Interface.Data.Reference.Time

This module defines an interface for `BusinessTime` rules, which are contracts to control and keep track of business time.

Interfaces

interface *Time*

An interface to manage and control business time. Controlled time rules (i.e. clocks) are managed by entities that have control certain business time events. These can be trading-open / -close on an exchange, start-of-day / end-of-day events of a trading desk, or just a daily clock tick to signal the passing of aticking. Intervals in which the clock "ticks" don't have to be regular, and can e.g. consider business days only.

viewtype *V*

Choice *Advance*
Advance time to its next state.
Controller: (DA.Internal.Record.getField @"providers" (view this))
Returns: (ContractId *Time*, ContractId *Event*)

Field	Type	Description
eventId	*Id*	Event identifier.
eventDescription	Text	Event description.

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Choice *Rewind*
Rewind time to its previous state.
Controller: (DA.Internal.Record.getField @"providers" (view this))
Returns: (ContractId *Time*, ContractId *Event*)

Field	Type	Description
eventId	*Id*	Event identifier.
eventDescription	Text	Event description.

Method advance : ContractId *Time* -> *Advance* -> Update (ContractId *Time*, ContractId *Event*)
Implementation of the `Advance` choice.
Method rewind : ContractId *Time* -> *Rewind* -> Update (ContractId *Time*, ContractId *Event*)
Implementation of the `Rewind` choice.

Data Types

type ***I*** = *Time*
Type synonym for `Time`.

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Time V*

data *View*

View for `Time`.

View

Field	Type	Description
providers	*Parties*	Parties controlling time.
id	*Id*	Textual identifier for the time rule.

instance Eq *View*

instance Show *View*

Functions

advance : *Time* -> ContractId *Time* -> *Advance* -> Update (ContractId *Time*, ContractId *Event*)

rewind : *Time* -> ContractId *Time* -> *Rewind* -> Update (ContractId *Time*, ContractId *Event*)

1.22.3.78 Daml.Finance.Interface.Holding.Base

Interfaces

interface *Base*

Base interface for a holding.

viewtype *V*

Choice *Acquire*
Lock a contract.
Controller: (DA.Internal.Record.getField @"owner" (DA.Internal.Record.getField @"account" (view this))), newLockers
Returns: ContractId *Base*

Field	Type	Description
newLockers	*Parties*	Parties which restrain the contract's ability to perform specified actions.
context	Text	Reason for acquiring a lock.
lockType	*LockType*	Type of lock to acquire

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *V*

Field	Type	Description
viewer	Party	The party retrieving the view.

Choice *Release*
Unlock a locked contract.
Controller: getLockers this
Returns: ContractId *Base*

Field	Type	Description
context	Text	

Method acquire : *Acquire* -> Update (ContractId *Base*)
Implementation of the `Acquire` choice.

Method release : *Release* -> Update (ContractId *Base*)
Implementation of the `Release` choice.

Data Types

type *I* = *Base*
Type synonym for `Base`.
instance HasMethod *Account* "credit" (*Credit* -> Update (ContractId *I*))
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))
instance HasMethod *Batch* "cancel" (*Cancel* -> Update [ContractId *I*])
instance HasMethod *Batch* "settle" (*Settle* -> Update [ContractId *I*])
instance HasMethod *Instruction* "allocate" (*Allocate* -> Update (ContractId *Instruction*, Optional (ContractId *I*)))
instance HasMethod *Instruction* "cancel" (*Cancel* -> Update (Optional (ContractId *I*)))
instance HasMethod *Instruction* "execute" (*Execute* -> Update (Optional (ContractId *I*)))

data *Lock*

Locking details.

Lock

Field	Type	Description
lockers	*Parties*	Parties which are locking the contract.
context	Set Text	Why this lock is held by the locking parties.
lockType	*LockType*	The type of lock applied.

instance Eq *Lock*

instance Show *Lock*

data *LockType*

Type of lock held.

Semaphore

A one time only lock.

Reentrant

A mutual exclusion lock where the same lockers may lock a contract multiple times.

instance Eq *LockType*

instance Show *LockType*

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Base* *V*

data *View*

View for `Base`.

View

Field	Type	Description
instrument	*InstrumentKey*	Instrument being held.
account	*AccountKey*	Key of the account holding the assets.
amount	Decimal	Size of the holding.
lock	Optional *Lock*	When a contract is locked, contains the locking details.

instance Eq *View*

instance Show *View*

Functions

acquire : *Base* -> *Acquire* -> Update (ContractId *Base*)

release : *Base* -> *Release* -> Update (ContractId *Base*)

getLockers : HasToInterface t *Base* => t -> *Parties*
Get the lockers of a holding.

1.22.3.79 Daml.Finance.Interface.Holding.Factory

Interfaces

interface *Factory*

Holding factory contract used to create (credit) holdings.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a holding on the instrument in the corresponding account.
Controller: (DA.Internal.Record.getField @"custodian" account), (DA.Internal.Record.getField @"owner" account)
Returns: ContractId *I*

Field	Type	Description
instrument	*InstrumentKey*	The instrument of which units are held.
account	*AccountKey*	The account at which the holding is held. Defines the holding's owner and custodian.
amount	Decimal	Number of units.
observers	*PartiesMap*	Observers of the holding to be credited.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.

Data Types

type ***F*** = *Factory*
Type synonym for `Factory`.

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

1.22.3.80 Daml.Finance.Interface.Holding.Fungible

Interfaces

interface *Fungible*

Interface for a fungible holding.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party fetching the view.

Choice *Merge*
Merge multiple fungible contracts into a single fungible contract.
Controller: (DA.Internal.Record.getField @"modifiers" (view this)), getLockers this
Returns: ContractId *Fungible*

Field	Type	Description
fungibleCids	[ContractId *Fungible*]	The fungible contracts to merge which will get consumed.

Choice *Split*
Split a fungible contract into multiple contracts by amount.
Controller: (DA.Internal.Record.getField @"modifiers" (view this)), getLockers this
Returns: *SplitResult*

Field	Type	Description
amounts	[Decimal]	The quantities to split the fungible asset by, creating a new contract per amount.

Method merge : *Merge* -> Update (ContractId *Fungible*)
Implementation of the `Merge` choice.
Method split : *Split* -> Update *SplitResult*
Implementation of the `Split` choice.

Data Types

type *I* = *Fungible*
Type synonym for `Fungible`.

data *SplitResult*

Result of a call to `Split`.

SplitResult

Field	Type	Description
splitCids	[ContractId *Fungible*]	The contract ids for the split holdings.
rest	Optional (ContractId *Fungible*)	Contract id for the holding on the remaining amount. It is `None` when the split is exact.

instance Eq *SplitResult*

instance Show *SplitResult*

instance HasMethod *Fungible* "split" (*Split* -> Update *SplitResult*)

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Fungible V*

data *View*

View for `Fungible`.

View

Field	Type	Description
modifiers	*Parties*	Parties which have the authorization to modify a fungible asset.

instance Eq *View*

instance Show *View*

Functions

split : *Fungible* -> *Split* -> Update *SplitResult*

merge : *Fungible* -> *Merge* -> Update (ContractId *Fungible*)

1.22.3.81 Daml.Finance.Interface.Holding.Transferable

Interfaces

interface *Transferable*

An interface respresenting a contract where ownership can be transferred to other parties.

viewtype *View*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Choice *Transfer*
Transfer a contract to a new owner.
Controller: actors, getLockers this
Returns: ContractId *Transferable*

Field	Type	Description
actors	*Parties*	Parties authorizing the transfer.
newOwnerAccount	*AccountKey*	The new owner's account.

Method transfer : ContractId *I* -> *Transfer* -> Update (ContractId *Transferable*)
Implementation of the `Transfer` choice.

Data Types

type *I* = *Transferable*
Type synonym for `Transferable`.
instance HasMethod *Transferable* "transfer" (ContractId *I* -> *Transfer* -> Update (ContractId *Transferable*))

type *V* = *View*
Type synonym for `View`.

data *View*

View for `Transferable`.

View

(no fields)

instance Eq *View*

instance Show *View*

instance HasFromAnyView *Transferable View*

Functions

transfer : *Transferable* -> ContractId *I* -> *Transfer* -> Update (ContractId *Transferable*)

1.22.3.82 Daml.Finance.Interface.Holding.Util

Functions

getInstrument : HasToInterface t *I* => t -> *InstrumentKey*
Get the key of a holding.

getAmount : HasToInterface t *I* => t -> Decimal
Get the amount of a holding.

disclose : (HasInterfaceTypeRep i, HasToInterface i *I*, HasFromInterface i *I*) => (Text, *Parties*) -> *Parties* -> ContractId i -> Update (ContractId i)
Disclose a holding.

undisclose : (HasInterfaceTypeRep i, HasToInterface i *I*, HasFromInterface i *I*) => (Text, *Parties*) -> *Parties* -> ContractId i -> Update (Optional (ContractId i))
Undisclose a holding.

1.22.3.83 Daml.Finance.Interface.Instrument.Base.Instrument

Interfaces

interface *Instrument*

Base interface for all instruments. This interface does not define any lifecycling logic.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *V*

Field	Type	Description
viewer	Party	The party retrieving the view.

Method getKey : *InstrumentKey*
Get the unique key for the `Instrument`.

Data Types

type ***I*** = *Instrument*
Type synonym for `Instrument`.
instance HasExerciseByKey Reference *InstrumentKey* GetCid (ContractId *I*)

type ***Q*** = *Quantity InstrumentKey* Decimal
Instrument quantity.

type ***R*** = Reference
Type synonym for `Reference`. This type is currently used as a work-around given the lack of interface keys.
instance HasExerciseByKey Reference *InstrumentKey* SetCid (ContractId *R*)
instance HasExerciseByKey Reference *InstrumentKey* SetObservers (ContractId *R*)

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Instrument V*

data *View*

View for `Instrument`.

View

Field	Type	Description
issuer	Party	The instrument's issuer.
depository	Party	The instrument's depository.
id	*Id*	The instrument's identifier.
version	Text	A textual instrument version.
description	Text	A human readable description of the instrument.
validAsOf	Time	Timestamp as of which the instrument is valid. This usually coincides with the timestamp of the event that creates the instrument. It usually does not coincide with ledger time. This is required for lifecycling of some instruments, in order to keep track of the last time the instrument was lifecycled. For instruments where this is not applicable, it can be set to the current time.

instance Eq *View*

instance Show *View*

Functions

getKey : *Instrument* -> *InstrumentKey*

exerciseInterfaceByKey : (HasInterfaceTypeRep i, HasExercise i d r) => *InstrumentKey* -> Party -> d -> Update r
Exercise interface by key. This method can be used to exercise a choice on an `Instrument` given its `InstrumentKey`. Requires as input the `InstrumentKey`, the actor fetching the instrument and the choice arguments. For example:

toKey : *V* -> *InstrumentKey*
Convert the instrument's View to its key.

fetchInstrument : HasToInterface t *I* => t -> Update *I*
Fetch instrument from holding.

qty : Decimal -> *InstrumentKey* -> *Q*
Wraps an amount and an instrument key into an instrument quantity.

scale : Decimal -> *Q* -> *Q*
Scale `Quantity` by the provided factor.

1.22.3.84 Daml.Finance.Interface.Instrument.Bond.Callable.Factory

Interfaces

interface *Factory*

Factory interface to instantiate callable bond instruments.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new instrument.
Controller: (DA.Internal.Record.getField @"depository" (DA.Internal.Record.getField @"instrument" callable)), (DA.Internal.Record.getField @"issuer" (DA.Internal.Record.getField @"instrument" callable))
Returns: ContractId *I*

Field	Type	Description
callable	*Callable*	Attributes to create a callable rate bond.
observers	*PartiesMap*	The instrument's observers.

Choice *Remove*
Archive an instrument.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ()

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.
Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View of `Factory`.

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.85 Daml.Finance.Interface.Instrument.Bond.Callable.Instrument

Interfaces

interface *Instrument*

Instrument interface representing a callable bond.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *V*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type ***I*** = *Instrument*
Type synonym for `Instrument`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Instrument V*

data *View*

View of `Instrument`.

View

Field	Type	Description
callable	*Callable*	Attributes of a callable bond.

instance Eq *View*

instance Show *View*

1.22.3.86 Daml.Finance.Interface.Instrument.Bond.Callable.Types

Data Types

data *Callable*

Describes the attributes of a Callable Bond.

Callable

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.
description	Text	The description of the bond.
floatingRate	Optional *FloatingRate*	A description of the floating rate to be used (if applicable).
couponRate	Decimal	The fixed coupon rate, per annum. For example, in case of a "3.5% p.a coupon" this should be 0.035. This can also used as a floating coupon spread. For example, in case of "3M Libor + 0.5%" this should be 0.005.
capRate	Optional Decimal	The maximum coupon rate possible. For example, if "3M Libor + 0.5%" would result in a rate of 2.5%, but capRate is 2.0%, the coupon rate used would be 2.0%.
floorRate	Optional Decimal	The minimum coupon rate possible. For example, if "3M Libor + 0.5%" would result in a rate of -0.2%, but floorRate is 0.0%, the coupon rate used would be 0.0%.
couponSchedule	*PeriodicSchedule*	The schedule for the periodic coupon payments. The coupon is paid on the last date of each schedule period. In case of a floating rate, the reference rate will be fixed in relation to this schedule (at the start/end of each period, as specified by FloatingRate). This is the main schedule of the instrument, which drives both the calculation and the payment of coupons. It also defines the issue date and the maturity date of the bond.
callSchedule	*PeriodicSchedule*	The bond is callable on the last date of each schedule period. For example, if this schedule is the same as the periodicSchedule, it means that the bond can be called on each coupon payment date.
noticeDays	Int	The number of business days in advance of the coupon date that the issuer must give notice if it wants to call the bond. The election whether to call or not to call must be done by this date.
holidayCalendarIds	[Text]	The identifiers of the holiday calendars to be used for the coupon schedule.
calendarDataProvider	Party	The reference data provider to use for the holiday calendar.
dayCountConvention	*DayCountConventionEnum*	The day count convention used to calculate day count fractions. For example: Act360.
useAdjustedDatesForDcf	Bool	Configure whether to use adjusted dates (as specified in *businessDayAdjustment* of the *couponSchedule*) for day count fractions.
currency	*InstrumentKey*	The currency of the bond. For example, if the bond pays in USD this should be a USD cash instrument.
notional	Decimal	The notional of the bond. This is the face

instance Eq *Callable*

instance Show *Callable*

1.22.3.87 Daml.Finance.Interface.Instrument.Bond.FixedRate.Factory

Interfaces

interface *Factory*

Factory interface to instantiate fixed-rate bond instruments.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new instrument.
Controller: (DA.Internal.Record.getField @"depository" (DA.Internal.Record.getField @"instrument" fixedRate)), (DA.Internal.Record.getField @"issuer" (DA.Internal.Record.getField @"instrument" fixedRate))
Returns: ContractId *I*

Field	Type	Description
fixedRate	*FixedRate*	Attributes to create a fixed rate bond.
observers	*PartiesMap*	The instrument's observers.

Choice *Remove*
Archive an instrument.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ()

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.
Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View of `Factory`.

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.88 Daml.Finance.Interface.Instrument.Bond.FixedRate.Instrument

Interfaces

interface *Instrument*

Instrument interface representing a fixed rate bond.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *V*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type *I* = *Instrument*
Type synonym for `Instrument`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Instrument V*

data *View*

View of `Instrument`.

View

Field	Type	Description
fixedRate	*FixedRate*	Attributes of a fixed rate bond.

instance Eq *View*

instance Show *View*

1.22.3.89 Daml.Finance.Interface.Instrument.Bond.FixedRate.Types

Data Types

data *FixedRate*

Describes the attributes of a Fixed Rate Bond.

FixedRate

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.
description	Text	The description of the bond.
couponRate	Decimal	The fixed coupon rate, per annum. For example, in case of a "3.5% p.a. coupon" this should be 0.035.
periodicSchedule	*PeriodicSchedule*	The schedule for the periodic coupon payments.
holidayCalendarIds	[Text]	The identifiers of the holiday calendars to be used for the coupon schedule.
calendarDataProvider	Party	The reference data provider to use for the holiday calendar.
dayCountConvention	*DayCountConventionEnum*	The day count convention used to calculate day count fractions. For example: Act360.
currency	*InstrumentKey*	The currency of the bond. For example, if the bond pays in USD this should be a USD cash instrument.
notional	Decimal	The notional of the bond. This is the face value corresponding to one unit of the bond instrument. For example, if one bond unit corresponds to 1000 USD, this should be 1000.0.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

instance Eq *FixedRate*

instance Show *FixedRate*

1.22.3.90 Daml.Finance.Interface.Instrument.Bond.FloatingRate.Factory

Interfaces

interface *Factory*

Factory interface to instantiate floating-rate bond instruments.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new instrument.
Controller: (DA.Internal.Record.getField @"depository" (DA.Internal.Record.getField @"instrument" floatingRate)), (DA.Internal.Record.getField @"issuer" (DA.Internal.Record.getField @"instrument" floatingRate))

Returns: ContractId *I*

Field	Type	Description
floatingRate	*FloatingRate*	Attributes to create a floating rate bond.
observers	*PartiesMap*	The instrument's observers.

Choice *Remove*
Archive an instrument.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ()

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.
Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View of `Factory`.

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.91 Daml.Finance.Interface.Instrument.Bond.FloatingRate.Instrument

Interfaces

interface *Instrument*

Instrument interface representing a floating rate bond.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *V*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type ***I*** = *Instrument*
Type synonym for `Instrument`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Instrument V*

data *View*

View of `Instrument`.

View

Field	Type	Description
floatingRate	*FloatingRate*	Attributes of a floating rate bond.

instance Eq *View*

instance Show *View*

1.22.3.92 Daml.Finance.Interface.Instrument.Bond.FloatingRate.Types

Data Types

data *FloatingRate*

Describes the attributes representing a floating rate bond.

FloatingRate

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.
description	Text	The description of the bond.
referenceRateId	Text	The floating rate reference ID. For example, in case of "3M Euribor + 0.5%" this should be a valid reference to the "3M Euribor" reference rate.
couponSpread	Decimal	The floating rate coupon spread. For example, in case of "3M Euribor + 0.5%" this should be 0.005.
periodicSchedule	*PeriodicSchedule*	The schedule for the periodic coupon payments.
holidayCalendarIds	[Text]	The identifiers of the holiday calendars to be used for the coupon schedule.
calendarDataProvider	Party	The reference data provider to use for the holiday calendar.
dayCountConvention	*DayCountConventionEnum*	The day count convention used to calculate day count fractions. For example: Act360.
currency	*InstrumentKey*	The currency of the bond. For example, if the bond pays in USD this should be a USD cash instrument.
notional	Decimal	The notional of the bond. This is the face value corresponding to one unit of the bond instrument. For example, if one bond unit corresponds to 1000 USD, this should be 1000.0.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

instance Eq *FloatingRate*

instance Show *FloatingRate*

1.22.3.93 Daml.Finance.Interface.Instrument.Bond.InflationLinked.Factory

Interfaces

interface *Factory*

Factory interface to instantiate inflation-linked bond instruments.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new instrument.
Controller: (DA.Internal.Record.getField @"depository" (DA.Internal.Record.getField @"instrument" inflationLinked)), (DA.Internal.Record.getField @"issuer" (DA.Internal.Record.getField @"instrument" inflationLinked))
Returns: ContractId *I*

Field	Type	Description
inflationLinked	*InflationLinked*	Attributes to create an inflation linked bond.
observers	*PartiesMap*	The instrument's observers.

Choice *Remove*
Archive an instrument.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ()

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.
Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View of `Factory`.

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.94 Daml.Finance.Interface.Instrument.Bond.InflationLinked.Instrument

Interfaces

interface *Instrument*

Instrument interface representing an inflation linked bond.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *V*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type ***I*** = *Instrument*
Type synonym for `Instrument`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Instrument V*

data *View*

View of `Instrument`.

View

Field	Type	Description
inflationLinked	*Inflation-Linked*	Attributes of an inflation linked bond.

instance Eq *View*

instance Show *View*

1.22.3.95 Daml.Finance.Interface.Instrument.Bond.InflationLinked.Types

Data Types

data *InflationLinked*

Describes the attributes of an Inflation Linked Bond.

InflationLinked

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.
description	Text	The description of the bond.
inflationIndexId	Text	The inflation index reference ID. For example, in case of "0.5% p.a. coupon, CPI adjusted principal" this should be a valid reference to the "CPI" index.
inflationIndexBaseValue	Decimal	The value of the inflation index on the first reference date of this bond (called "dated date" on US TIPS). This is used as the base value for the principal adjustment.
couponRate	Decimal	The fixed coupon rate, per annum. For example, in case of a "0.5% p.a. coupon, CPI adjusted principal" this should be 0.005.
periodicSchedule	*PeriodicSchedule*	The schedule for the periodic coupon payments.
holidayCalendarIds	[Text]	The identifiers of the holiday calendars to be used for the coupon schedule.
calendarDataProvider	Party	The reference data provider to use for the holiday calendar.
dayCountConvention	*DayCountConventionEnum*	The day count convention used to calculate day count fractions. For example: Act360.
currency	*InstrumentKey*	The currency of the bond. For example, if the bond pays in USD this should be a USD cash instrument.
notional	Decimal	The notional of the bond. This is the face value corresponding to one unit of the bond instrument. For example, if one bond unit corresponds to 1000 USD, this should be 1000.0.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

instance Eq *InflationLinked*

instance Show *InflationLinked*

1.22.3.96 Daml.Finance.Interface.Instrument.Bond.ZeroCoupon.Factory

Interfaces

interface *Factory*

Factory interface to instantiate zero-coupon bond instruments.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new instrument.
Controller: (DA.Internal.Record.getField @"depository" (DA.Internal.Record.getField @"instrument" zeroCoupon)), (DA.Internal.Record.getField @"issuer" (DA.Internal.Record.getField @"instrument" zeroCoupon))
Returns: ContractId *I*

Field	Type	Description
zeroCoupon	*ZeroCoupon*	Attributes to create a zero coupon bond.
observers	*PartiesMap*	The instrument's observers.

Choice *Remove*
Archive an instrument.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ()

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.
Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View of `Factory`.

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.97 Daml.Finance.Interface.Instrument.Bond.ZeroCoupon.Instrument

Interfaces

interface *Instrument*

Instrument interface representing a zero coupon bond.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *V*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type ***I*** = *Instrument*
Type synonym for `Instrument`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Instrument V*

data *View*

View of `Instrument`.

View

Field	Type	Description
zeroCoupon	*ZeroCoupon*	Attributes of a zero coupon bond.

instance Eq *View*

instance Show *View*

1.22.3.98 Daml.Finance.Interface.Instrument.Bond.ZeroCoupon.Types

Data Types

data *ZeroCoupon*

Describes the attributes of a Zero Coupon bond.

ZeroCoupon

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.
description	Text	The description of the bond.
issueDate	Date	The date when the bond was issued.
maturityDate	Date	The redemption date of the bond.
currency	*InstrumentKey*	The currency of the bond. For example, if the bond pays in USD this should be a USD cash instrument.
notional	Decimal	The notional of the bond. This is the face value corresponding to one unit of the bond instrument. For example, if one bond unit corresponds to 1000 USD, this should be 1000.0.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

instance Eq *ZeroCoupon*

instance Show *ZeroCoupon*

1.22.3.99 Daml.Finance.Interface.Instrument.Bond.Types

Data Types

data *DateRelativeToEnum*

The specification of whether payments/resets occur relative to the first or last day of a calculation period.

CalculationPeriodStartDate

Payments/Resets will occur relative to the first day of each calculation period.

CalculationPeriodEndDate

Payments/Resets will occur relative to the last day of each calculation period.

instance Eq *DateRelativeToEnum*

instance Show *DateRelativeToEnum*

data *DayTypeEnum*

A day type classification used in counting the number of days between two dates.

Business

When calculating the number of days between two dates the count includes only business days.

Calendar

When calculating the number of days between two dates the count includes all calendar days.

instance Eq *DayTypeEnum*

instance Show *DayTypeEnum*

data *FixingDates*

Specifies the fixing date relative to the reset date in terms of a business days offset and an associated set of financial business centers.

FixingDates

Field	Type	Description
period	*PeriodEnum*	The unit of the date offset, e.g. D means that the date offset is specified in days.
periodMultiplier	Int	The number of days (if period is D) before or after the base date the fixing is observed.
dayType	Optional *DayTypeEnum*	Indicate whether the date offset is given in Business days or Calendar days.
businessDayConvention	*BusinessDayConventionEnum*	Business day convention that describes how a non-business day is adjusted.
businessCenters	[Text]	The identifiers of the holiday calendars to be used for date adjustment (if any).

instance Eq *FixingDates*

instance Show *FixingDates*

data *FloatingRate*

Specifies the data required for a floating rate coupon.

FloatingRate

Field	Type	Description
referenceRateId	Text	The identifier of the reference rate to be used for the coupon, e.g. Libor-3M.
referenceRateType	*ReferenceRateTypeEnum*	The type of reference rate, which defines how the reference rate is calcuated.
fixingDates	*FixingDates*	Specifies the fixing dates as an offset of the calculation date, e.g. -2 business days.

instance Eq *FloatingRate*

instance Show *FloatingRate*

data *ReferenceRateTypeEnum*

The type of reference rate, which defines how the reference rate is calcuated.

SingleFixing DateRelativeToEnum

The reference rate is fixed on one observation date. This is usually the case for Libor and similar reference rates. A DateRelativeToEnum is required to indicate whether the reference rate will reset relative to the first or the last day of the calculation period.

CompoundedIndex DayCountConventionEnum

The reference rate is a regularly (e.g. daily) compounded reference rate, e.g. compounded SOFR, calculated via an index that is continuously compounded since

a specified start date. This enables efficient calculation using only the index values at the start and at the end of the calculation period: SOFR_INDEX_END / SOFR_INDEX_START - 1, as described here: https://www.newyorkfed.org/markets/reference-rates/additional-information-about-reference-rates#tgcr_bgcr_sofr_calculation_
The day count convention used for the index calculation (by the index provider) is also required. For example, in the case of SOFR this is Act360, which is implied by the 360/dc factor in the formula in the "Calculation Methodology for the SOFR Averages and Index" section in the link above.

instance Eq *ReferenceRateTypeEnum*

instance Show *ReferenceRateTypeEnum*

1.22.3.100 Daml.Finance.Interface.Instrument.Equity.Factory

Interfaces

interface *Factory*

Factory interface to instantiate equities.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new instrument.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ContractId *I*

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.
description	Text	A description of the instrument.
validAsOf	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.
observers	*PartiesMap*	The instrument's observers.

Choice *Remove*
Archive an instrument.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ()

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.

Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type ***F*** = *Factory*
Type synonym for `Factory`.

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.101 Daml.Finance.Interface.Instrument.Equity.Instrument

Interfaces

interface *Instrument*

An interface for a generic equity instrument.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *DeclareDistribution*
Declare a distribution (e.g. a dividend or a rights issue) to shareholders.
Controller: (DA.Internal.Record.getField @"issuer" (view $ toInterface @BaseInstrument.I this))
Returns: ContractId *I*

Field	Type	Description
id	*Id*	Event identifier of the dividend distribution.
description	Text	Description of the dividend event.
effectiveTime	Time	Time at which the dividend is distributed.
newInstrument	*InstrumentKey*	Instrument held after the dividend distribution (i.e. "ex-dividend" stock).
perUnitDistribution	[*InstrumentQuantity*]	Distributed quantities per unit held.

Choice *DeclareReplacement*
Declare a replacement event, where units of the instrument are replaced by a basket of other instruments.
Controller: (DA.Internal.Record.getField @"issuer" (view $ toInterface @BaseInstrument.I this))
Returns: ContractId *I*

Field	Type	Description
id	*Id*	Distribution Id.
description	Text	Description of the replacement event.
effectiveTime	Time	Time the replacement is to be executed.
perUnitReplacement	[*InstrumentQuantity*]	Payout offered to shareholders per held share.

Choice *DeclareStockSplit*
Declare a stock split.
Controller: (DA.Internal.Record.getField @"issuer" (view $ toInterface @BaseInstrument.I this))
Returns: ContractId *I*

Field	Type	Description
id	*Id*	Event identifier of the stock split.
description	Text	Description of the stock split event.
effectiveTime	Time	Time at which the stock split is effective.
newInstrument	*InstrumentKey*	Instrument to be held after the stock split is executed.
adjustmentFactor	Decimal	Adjustment factor for the stock split.

Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Method declareDistribution : *DeclareDistribution* -> Update (ContractId *I*)
Implementation for the `DeclareDistribution` choice.
Method declareReplacement : *DeclareReplacement* -> Update (ContractId *I*)
Implementation for the `DeclareReplacement` choice.
Method declareStockSplit : *DeclareStockSplit* -> Update (ContractId *I*)
Implementation for the `DeclareStockSplit` choice.

Data Types

type *I* = *Instrument*

Type synonym for `Instrument`.

instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type *V* = *View*

Type synonym for `View`.

instance HasFromAnyView *Instrument V*

data *View*

View for `Instrument`.

View

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.

instance Eq *View*

instance Show *View*

Functions

declareDistribution : *Instrument* -> *DeclareDistribution* -> Update (ContractId *I*)

declareStockSplit : *Instrument* -> *DeclareStockSplit* -> Update (ContractId *I*)

declareReplacement : *Instrument* -> *DeclareReplacement* -> Update (ContractId *I*)

1.22.3.102 Daml.Finance.Interface.Instrument.Generic.Factory

Interfaces

interface *Factory*

Factory interface to instantiate generic instruments using Contingent Claims.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new generic instrument.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ContractId *I*

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.
description	Text	A description of the instrument.
claims	*C*	The claim tree.
acquisitionTime	Time	The claim's acquisition time. This usually corresponds to the start date of the contract.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.
observers	*PartiesMap*	The instrument's observers.

Choice *Remove*
Archive a generic instrument.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ()

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.
Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type ***F*** = *Factory*
Type synonym for `Factory`.

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.103 Daml.Finance.Interface.Instrument.Generic.Instrument

Interfaces

interface *Instrument*

Interface for generic instruments utilizing Contingent Claims.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *V*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type ***I*** = *Instrument*
Type synonym for `Instrument`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Instrument V*

data *View*

View

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.
claims	*C*	The claim tree.

instance Eq *View*

instance Show *View*

1.22.3.104 Daml.Finance.Interface.Instrument.Option.BarrierEuropeanCash.Factory

Interfaces

interface *Factory*

Factory interface to instantiate barrier options.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new instrument.
Controller: (DA.Internal.Record.getField @"depository" (DA.Internal.Record.getField @"instrument" barrierEuropean)), (DA.Internal.Record.getField @"issuer" (DA.Internal.Record.getField @"instrument" barrierEuropean))
Returns: ContractId *I*

Field	Type	Description
barrierEuropean	*BarrierEuropean*	Attributes to create a barrier option.
observers	*PartiesMap*	The instrument's observers.

Choice *Remove*
Archive an instrument.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ()

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.
Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View of `Factory`.

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.105 Daml.Finance.Interface.Instrument.Option.BarrierEuropeanCash.Instrument

Interfaces

interface *Instrument*

Instrument interface representing a barrier option.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *V*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type ***I*** = *Instrument*
Type synonym for `Instrument`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Instrument V*

data *View*

View of `Instrument`.

View

Field	Type	Description
barrierEuropean	*BarrierEuropean*	Attributes of a barrier option.

instance Eq *View*

instance Show *View*

1.22.3.106 Daml.Finance.Interface.Instrument.Option.BarrierEuropeanCash.Types

Data Types

data *BarrierEuropean*

Describes the attributes of a cash-settled barrier option with European exercise.

BarrierEuropean

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.
description	Text	The description of the option.
referenceAssetId	Text	The reference asset ID. For example, in case of an option on AAPL this should be a valid reference to the AAPL fixings to be used for the payoff calculation.
ownerReceives	Bool	Indicate whether a holding owner of this instrument receives the option payoff.
optionType	*OptionTypeEnum*	Indicate whether the option is a call or a put.
strike	Decimal	The strike price of the option.
barrier	Decimal	The barrier price of the option.
barrierType	*BarrierTypeEnum*	The type of barrier.
barrierStartDate	Date	The start date for barrier observations.
expiryDate	Date	The expiry date of the option.
currency	*InstrumentKey*	The currency of the option. For example, if the option pays in USD this should be a USD cash instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

instance Eq *BarrierEuropean*

instance Show *BarrierEuropean*

1.22.3.107 Daml.Finance.Interface.Instrument.Option.Dividend.Factory

Interfaces

interface *Factory*

Factory interface to instantiate Dividend options.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new instrument.
Controller: (DA.Internal.Record.getField @"depository" (DA.Internal.Record.getField @"instrument" dividend)), (DA.Internal.Record.getField @"issuer" (DA.Internal.Record.getField @"instrument" dividend))
Returns: ContractId *I*

Field	Type	Description
dividend	*Dividend*	Attributes to create a Dividend option.
observers	*PartiesMap*	The instrument's observers.

Choice *Remove*
Archive an instrument.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ()

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.
Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View of `Factory`.

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.108 Daml.Finance.Interface.Instrument.Option.Dividend.Instrument

Interfaces

interface *Instrument*

Instrument interface representing a physically settled Dividend option.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *V*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type ***I*** = *Instrument*
Type synonym for `Instrument`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Instrument V*

data *View*

View of `Instrument`.

View

Field	Type	Description
dividend	*Dividend*	Attributes of a Dividend option.

instance Eq *View*

instance Show *View*

1.22.3.109 Daml.Finance.Interface.Instrument.Option.Dividend.Types

Data Types

data *Dividend*

Describes the attributes of a physically settled Dividend option.

Dividend

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.
description	Text	The description of the option.
expiryDate	Date	The expiry date of the option.
cashQuantity	*InstrumentQuantity*	Dividend paid in cash
sharesQuantity	Optional *InstrumentQuantity*	Dividend paid in shares (if applicable)
fxQuantity	Optional *InstrumentQuantity*	Dividend paid in a foreign currency (if applicable)
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.
prevElections	[EventData]	A list of previous elections that have been lifecycled on this instrument so far.

instance Eq *Dividend*

instance Show *Dividend*

data *ElectionTypeEnum*

An election type classification.

Shares

Shares dividend.

Cash

Cash dividend.

CashFx

Foreign currency cash dividend.

instance Eq *ElectionTypeEnum*

instance Show *ElectionTypeEnum*

1.22.3.110 Daml.Finance.Interface.Instrument.Option.Dividend.Election.Factory

Interfaces

interface *Factory*

Factory interface to instantiate elections on generic instruments.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new Election.
Controller: actors
Returns: ContractId *I*

Field	Type	Description
actors	*Parties*	Parties calling the `Create` choice.
elector	Party	Parties on behalf of which the election is made.
counterparty	Party	Faces the `elector` in the `Holding`.
provider	Party	Party that signs the election (together with the elector).
id	*Id*	The identifier for an election.
description	Text	A description of the instrument.
claimType	*ElectionType-Enum*	The election type corresponding to the elected sub-tree.
electorIsOwner	Bool	`True` if election is on behalf of the owner of the holding, `False` otherwise.
electionTime	Time	Time at which the election is put forward.
observers	*PartiesMap*	Observers of the election.
amount	Decimal	Number of instrument units to which the election applies.
instrument	*Instrumen-tKey*	The instrument to which the election applies.

Choice *Remove*
Archive an Election.
Controller: actors
Returns: ()

Field	Type	Description
actors	*Parties*	Parties executing the `Remove` choice.
electionCid	ContractId *I*	The election's contract id.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.
Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type ***F*** = *Factory*
Type synonym for `Factory`.

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.111 Daml.Finance.Interface.Instrument.Option.EuropeanCash.Factory

Interfaces

interface *Factory*

Factory interface to instantiate European options.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)

Choice *Create*
Create a new instrument.
Controller: (DA.Internal.Record.getField @"depository" (DA.Internal.Record.getField @"instrument" european)), (DA.Internal.Record.getField @"issuer" (DA.Internal.Record.getField @"instrument" european))
Returns: ContractId *I*

Field	Type	Description
european	*European*	Attributes to create a European option.
observers	*PartiesMap*	The instrument's observers.

Choice *Remove*
Archive an instrument.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ()

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.
Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type ***F*** = *Factory*
Type synonym for `Factory`.

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View of `Factory`.

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.112 Daml.Finance.Interface.Instrument.Option.EuropeanCash.Instrument

Interfaces

interface *Instrument*

Instrument interface representing a European option.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *V*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type ***I*** = *Instrument*
Type synonym for `Instrument`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Instrument V*

data *View*

View of `Instrument`.

View

Field	Type	Description
european	*European*	Attributes of a European option.

instance Eq *View*

instance Show *View*

1.22.3.113 Daml.Finance.Interface.Instrument.Option.EuropeanCash.Types

Data Types

data *European*

Describes the attributes of a cash-settled European option.

European

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.
description	Text	The description of the option.
referenceAssetId	Text	The reference asset ID. For example, in case of an option on AAPL this should be a valid reference to the AAPL fixings to be used for the payoff calculation.
ownerReceives	Bool	Indicate whether a holding owner of this instrument receives the option payoff.
optionType	*OptionTypeEnum*	Indicate whether the option is a call or a put.
strike	Decimal	The strike price of the option.
expiryDate	Date	The expiry date of the option.
currency	*InstrumentKey*	The currency of the option. For example, if the option pays in USD this should be a USD cash instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

instance Eq *European*

instance Show *European*

1.22.3.114 Daml.Finance.Interface.Instrument.Option.EuropeanPhysical.Factory

Interfaces

interface *Factory*

Factory interface to instantiate European options.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new instrument.

Controller: (DA.Internal.Record.getField @"depository" (DA.Internal.Record.getField @"instrument" european)), (DA.Internal.Record.getField @"issuer" (DA.Internal.Record.getField @"instrument" european))
Returns: ContractId *I*

Field	Type	Description
european	*European*	Attributes to create a European option.
observers	*PartiesMap*	The instrument's observers.

Choice *Remove*
Archive an instrument.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ()

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.
Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type ***F*** = *Factory*
Type synonym for `Factory`.

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View of `Factory`.

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.115 Daml.Finance.Interface.Instrument.Option.EuropeanPhysical.Instrument

Interfaces

interface *Instrument*

Instrument interface representing a physically settled European option.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *V*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type ***I*** = *Instrument*
Type synonym for `Instrument`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Instrument V*

data *View*

View of `Instrument`.

View

Field	Type	Description
european	*European*	Attributes of a European option.

instance Eq *View*

instance Show *View*

1.22.3.116 Daml.Finance.Interface.Instrument.Option.EuropeanPhysical.Types

Data Types

data *European*

Describes the attributes of a physically settled European option.

European

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.
description	Text	The description of the option.
referenceAsset	*InstrumentKey*	The reference asset. For example, in case of an option on AAPL this should be an AAPL instrument.
ownerReceives	Bool	Indicate whether a holding owner of this instrument receives the option payoff.
optionType	*OptionTypeEnum*	Indicate whether the option is a call or a put.
strike	Decimal	The strike price of the option.
expiryDate	Date	The expiry date of the option.
currency	*InstrumentKey*	The currency of the option. For example, if the option pays in USD this should be a USD cash instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.
prevElections	[EventData]	A list of previous elections that have been lifecycled on this instrument so far.

instance Eq *European*

instance Show *European*

1.22.3.117 Daml.Finance.Interface.Instrument.Option.Types

Data Types

data *BarrierTypeEnum*

A barrier type classification.

UpAndOut

The option is knocked out if the underlying trades at or above the barrier.

DownAndOut

The option is knocked out if the underlying trades at or below the barrier.

UpAndIn

The option is activated if the underlying trades at or above the barrier.

DownAndIn

The option is activated if the underlying trades at or below the barrier.

instance Eq *BarrierTypeEnum*

instance Show *BarrierTypeEnum*

data *OptionTypeEnum*

An option type classification.

Call

Call option.

Put

Put option.

instance Eq *OptionTypeEnum*

instance Show *OptionTypeEnum*

1.22.3.118 Daml.Finance.Interface.Instrument.Swap.Asset.Factory

Interfaces

interface *Factory*

Factory interface to instantiate asset swaps.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new instrument.
Controller: (DA.Internal.Record.getField @"depository" (DA.Internal.Record.getField @"instrument" asset)), (DA.Internal.Record.getField @"issuer" (DA.Internal.Record.getField @"instrument" asset))
Returns: ContractId *I*

Field	Type	Description
asset	*Asset*	Attributes to create an asset swap.
observers	*PartiesMap*	The instrument's observers.

Choice *Remove*
Archive an instrument.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ()

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.
Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type ***F*** = *Factory*
Type synonym for `Factory`.

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View of `Factory`.

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.119 Daml.Finance.Interface.Instrument.Swap.Asset.Instrument

Interfaces

interface *Instrument*

Instrument interface representing an asset swap.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)

Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *V*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type *I* = *Instrument*
Type synonym for `Instrument`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Instrument V*

data *View*

View of `Instrument`.

View

Field	Type	Description
asset	*Asset*	Attributes of an asset swap.

instance Eq *View*

instance Show *View*

1.22.3.120 Daml.Finance.Interface.Instrument.Swap.Asset.Types

Data Types

data *Asset*

Describes the attributes of an Asset swap.

Asset

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.
description	Text	The description of the swap.
referenceAssetId	Text	The reference asset ID. For example, in case of "AAPL total return vs 2.5% fix" this should be a valid reference to the AAPL fixings to be used for the total return calculation (dividend-adjusted fixings).
ownerReceivesFix	Bool	Indicate whether a holding owner of this instrument receives the fix or the asset leg of the swap.
fixRate	Decimal	The interest rate of the fix leg. For example, in case of "AAPL total return vs 2.5% fix" this should be 0.025.
periodicSchedule	*PeriodicSchedule*	The schedule for the periodic swap payments.
holidayCalendarIds	[Text]	The identifiers of the holiday calendars to be used for the swap payment schedule.
calendarDataProvider	Party	The reference data provider to use for the holiday calendar.
dayCountConvention	*DayCountConventionEnum*	The day count convention used to calculate day count fractions. For example: Act360.
currency	*InstrumentKey*	The currency of the swap. For example, if the swap pays in USD this should be a USD cash instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

instance Eq *Asset*

instance Show *Asset*

1.22.3.121 Daml.Finance.Interface.Instrument.Swap.CreditDefault.Factory

Interfaces

interface *Factory*

Factory interface to instantiate credit default swaps.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)

Choice *Create*
Create a new instrument.
Controller: (DA.Internal.Record.getField @"depository" (DA.Internal.Record.getField @"instrument" creditDefault)), (DA.Internal.Record.getField @"issuer" (DA.Internal.Record.getField @"instrument" creditDefault))
Returns: ContractId *I*

Field	Type	Description
creditDefault	*CreditDefault*	Attributes to create a credit default swap.
observers	*PartiesMap*	The instrument's observers.

Choice *Remove*
Archive an instrument.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ()

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.
Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type ***F*** = *Factory*
Type synonym for `Factory`.

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View of `Factory`.

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.122 Daml.Finance.Interface.Instrument.Swap.CreditDefault.Instrument

Interfaces

interface *Instrument*

Instrument interface representing a credit default swap.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *V*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type ***I*** = *Instrument*
Type synonym for `Instrument`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Instrument V*

data *View*

View of `Instrument`.

View

Field	Type	Description
creditDefault	*CreditDefault*	Attributes of a credit default swap.

instance Eq *View*

instance Show *View*

1.22.3.123 Daml.Finance.Interface.Instrument.Swap.CreditDefault.Types

Data Types

data *CreditDefault*

Describes the attributes of a Credit Default swap.

CreditDefault

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.
description	Text	The description of the swap.
defaultProbabilityReferenceId	Text	The reference ID of the default probability observable. For example, in case of protection against a "TSLA bond payment default" this should be a valid reference to the "TSLA default probability".
recoveryRateReferenceId	Text	The reference ID of the recovery rate observable. For example, in case of a "TSLA bond payment default with a 60% recovery rate" this should be a valid reference to the "TSLA bond recovery rate".
ownerReceivesFix	Bool	Indicate whether a holding owner of this instrument receives the fix or the default protection leg of the swap.
fixRate	Decimal	The interest rate of the fix leg. For example, in case of "2.5% fix" this should be 0.025.
periodicSchedule	*PeriodicSchedule*	The schedule for the periodic swap payments.
holidayCalendarIds	[Text]	The identifiers of the holiday calendars to be used for the swap payment schedule.
calendarDataProvider	Party	The reference data provider to use for the holiday calendar.
dayCountConvention	*DayCountConventionEnum*	The day count convention used to calculate day count fractions. For example: Act360.
currency	*InstrumentKey*	The currency of the swap. For example, if the swap pays in USD this should be a USD cash instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

instance Eq *CreditDefault*

instance Show *CreditDefault*

1.22.3.124 Daml.Finance.Interface.Instrument.Swap.Currency.Factory

Interfaces

interface *Factory*

Factory interface to instantiate currency swaps.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new instrument.
Controller: (DA.Internal.Record.getField @"depository" (DA.Internal.Record.getField @"instrument" currencySwap)), (DA.Internal.Record.getField @"issuer" (DA.Internal.Record.getField @"instrument" currencySwap))
Returns: ContractId *I*

Field	Type	Description
currencySwap	*CurrencySwap*	Attributes to create a currency swap.
observers	*PartiesMap*	The instrument's observers.

Choice *Remove*
Archive an instrument.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ()

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.
Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View of `Factory`.

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.125 Daml.Finance.Interface.Instrument.Swap.Currency.Instrument

Interfaces

interface *Instrument*

Instrument interface representing a currency swap.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *V*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type ***I*** = *Instrument*
Type synonym for `Instrument`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Instrument V*

data *View*

View of `Instrument`.

View

Field	Type	Description
currencySwap	*CurrencySwap*	Attributes of a currency swap.

instance Eq *View*

instance Show *View*

1.22.3.126 Daml.Finance.Interface.Instrument.Swap.Currency.Types

Data Types

data *CurrencySwap*

Describes the attributes of a Currency swap.

CurrencySwap

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.
description	Text	The description of the swap.
ownerReceivesBase	Bool	Indicate whether a holding owner of this instrument receives the base currency leg or the foreign currency leg of the swap.
baseRate	Decimal	The interest rate of the base currency. For example, in case of "3% in USD" this should be 0.03.
foreignRate	Decimal	The interest rate of the foreign currency. For example, in case of "2% in EUR" this should be 0.02.
periodicSchedule	*PeriodicSchedule*	The schedule for the periodic swap payments.
holidayCalendarIds	[Text]	The identifiers of the holiday calendars to be used for the swap payment schedule.
calendarDataProvider	Party	The reference data provider to use for the holiday calendar.
dayCountConvention	*DayCountConventionEnum*	The day count convention used to calculate day count fractions. For example: Act360.
baseCurrency	*InstrumentKey*	The base currency of the swap. For example, in the case of USD this should be a USD cash instrument.
foreignCurrency	*InstrumentKey*	The foreign currency of the swap. For example, in case of EUR this should be a EUR cash instrument.
fxRate	Decimal	The fx rate used to convert from the base currency principal amount to the foreign currency principal amount.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

instance Eq *CurrencySwap*

instance Show *CurrencySwap*

1.22.3.127 Daml.Finance.Interface.Instrument.Swap.ForeignExchange.Factory

Interfaces

interface *Factory*

Factory interface to instantiate foreign exchange swaps.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new instrument.
Controller: (DA.Internal.Record.getField @"depository" (DA.Internal.Record.getField @"instrument" foreignExchange)), (DA.Internal.Record.getField @"issuer" (DA.Internal.Record.getField @"instrument" foreignExchange))
Returns: ContractId *I*

Field	Type	Description
foreignExchange	*ForeignExchange*	Attributes to create an FX swap.
observers	*PartiesMap*	The instrument's observers.

Choice *Remove*
Archive an instrument.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ()

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.
Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View of `Factory`.

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.128 Daml.Finance.Interface.Instrument.Swap.ForeignExchange.Instrument

Interfaces

interface *Instrument*

Instrument interface representing an FX swap.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *V*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type ***I*** = *Instrument*
Type synonym for `Instrument`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Instrument V*

data *View*

View of `Instrument`.

View

Field	Type	Description
foreignExchange	*ForeignExchange*	Attributes of an FX swap.

instance Eq *View*

instance Show *View*

1.22.3.129 Daml.Finance.Interface.Instrument.Swap.ForeignExchange.Types

Data Types

data *ForeignExchange*

Describes the attributes of a Foreign Exchange swap.

ForeignExchange

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.
description	Text	The description of the swap.
firstFxRate	Decimal	The fx rate used for the first swap payment.
finalFxRate	Decimal	The fx rate used for the final swap payment.
issueDate	Date	The date when the swap was issued.
firstPaymentDate	Date	The first payment date of the swap.
maturityDate	Date	The final payment date of the swap.
baseCurrency	*InstrumentKey*	The base currency of the swap, which will be exchanged to another (foreign) currency on the first payment date. For example, in case of USD this should be a USD cash instrument.
foreignCurrency	*InstrumentKey*	The foreign currency of the swap. For example, in case of EUR this should be a EUR cash instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

instance Eq *ForeignExchange*

instance Show *ForeignExchange*

1.22.3.130 Daml.Finance.Interface.Instrument.Swap.Fpml.Factory

Interfaces

interface *Factory*

Factory interface to instantiate FpML swaps.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new instrument.
Controller: (DA.Internal.Record.getField @"depository" (DA.Internal.Record.getField @"instrument" fpml)), (DA.Internal.Record.getField @"issuer" (DA.Internal.Record.getField @"instrument" fpml))
Returns: ContractId *I*

Field	Type	Description
fpml	*Fpml*	Attributes to create a swap specified as FpML swapStreams.
observers	*PartiesMap*	The instrument's observers.

Choice *Remove*
Archive an instrument.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ()

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.
Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View of `Factory`.

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.131 Daml.Finance.Interface.Instrument.Swap.Fpml.FpmlTypes

Data Types

data *AdjustableDate*

A type for defining a date that shall be subject to adjustment if it would otherwise fall on a day that is not a business day in the specified business centers, together with the convention for adjusting the date.

AdjustableDate

Field	Type	Description
unadjustedDate	Date	A date subject to adjustment.
dateAdjustments	*BusinessDayAdjustments*	The business day convention and financial business centers used for adjusting the date if it would otherwise fall on a day that is not a business date in the specified business centers. adjustedDate : Optional IdentifiedDate ^ The date once the adjustment has been performed. (Note that this date may change if the business center holidays change).

instance Eq *AdjustableDate*

instance Show *AdjustableDate*

data *BusinessCenterTime*

A type for defining a time with respect to a business day calendar location. For example, 11:00am London time.

BusinessCenterTime

Field	Type	Description
hourMinuteTime	*HourMinuteTime*	A time specified in hh:mm:ss format where the second component must be '00', e.g., 11am would be represented as 11:00:00.
businessCenter	Text	

instance Eq *BusinessCenterTime*

instance Show *BusinessCenterTime*

data *BusinessDayAdjustments*

A type defining the business day convention and financial business centers used for adjusting any relevant date if it would otherwise fall on a day that is not a business day in the specified business centers.

BusinessDayAdjustments

Field	Type	Description
businessDayConvention	*BusinessDayConventionEnum*	The convention for adjusting a date if it would otherwise fall on a day that is not a business day.
businessCenters	[Text]	

instance Eq *BusinessDayAdjustments*

instance Show *BusinessDayAdjustments*

data *Calculation*

The parameters used in the calculation of fixed or floating rate period amounts.

Calculation

Field	Type	Description
notionalScheduleValue	*NotionalScheduleValue*	
rateTypeValue	*RateTypeValue*	
dayCountFraction	*DayCountConventionEnum*	
compoundingMethodEnum	Optional *CompoundingMethodEnum*	

instance Eq *Calculation*

instance Show *Calculation*

data *CalculationPeriodAmount*

The calculation period amount parameters.

CalculationPeriodAmount

Field	Type	Description
calculation	*Calculation*	

instance Eq *CalculationPeriodAmount*

instance Show *CalculationPeriodAmount*

data *CalculationPeriodDates*

The calculation periods dates schedule.

CalculationPeriodDates

Field	Type	Description
id	Text	
effectiveDate	*AdjustableDate*	
terminationDate	*AdjustableDate*	
calculationPeriodDatesAdjustments	*CalculationPeriodDatesAdjustments*	
firstPeriodStartDate	Optional *AdjustableDate*	
firstRegularPeriodStartDate	Optional Date	
lastRegularPeriodEndDate	Optional Date	
calculationPeriodFrequency	*CalculationPeriodFrequency*	

instance Eq *CalculationPeriodDates*

instance Show *CalculationPeriodDates*

data *CalculationPeriodDatesAdjustments*

The business day convention to apply to each calculation period end date if it would otherwise fall on a day that is not a business day in the specified financial business centers.

CalculationPeriodDatesAdjustments

Field	Type	Description
businessDayConvention	*BusinessDayConventionEnum*	
businessCenters	[Text]	

instance Eq *CalculationPeriodDatesAdjustments*

instance Show *CalculationPeriodDatesAdjustments*

data *CalculationPeriodFrequency*

A type defining the frequency at which calculation period end dates occur within the regular part of the calculation period schedule and thier roll date convention. In case the calculation frequency is of value T (term), the period is defined by the swap\swapStream\calculationPerioDates\effectiveDate and the swap\swapStream\calculationPerioDates\terminationDate.

CalculationPeriodFrequency

Field	Type	Description
periodMultiplier	Int	A time period multiplier, e.g., 1, 2 or 3 etc. If the period value is T (Term) then periodMultiplier must contain the value 1.
period	*PeriodExtendedEnum*	A time period, e.g., a day, week, month, year or term of the stream.
rollConvention	*RollConventionEnum*	Used in conjunction with a frequency and the regular period start date of a calculation period, determines each calculation period end date within the regular part of a c alculation period schedule.

instance Eq *CalculationPeriodFrequency*

instance Show *CalculationPeriodFrequency*

data *CompoundingMethodEnum*

The compounding calculation method

Flat

Flat compounding. Compounding excludes the spread. Note that the first compounding period has it's interest calculated including any spread then subsequent periods compound this at a rate excluding the spread.

NoCompounding

No compounding is to be applied.

Straight

Straight compounding. Compounding includes the spread.

SpreadExclusive

Spread Exclusive compounding.

instance Eq *CompoundingMethodEnum*

instance Show *CompoundingMethodEnum*

data *DateOffset*

A type defining an offset used in calculating a date when this date is defined in reference to another date through a date offset. The type includes the convention for adjusting the date and an optional sequence element to indicate the order in a sequence of multiple date offsets.

DateOffset

Field	Type	Description
periodMultiplier	Int	A time period multiplier, e.g. 1, 2 or 3 etc. A negative value can be used when specifying an offset relative to another date, e.g. -2 days.
period	*PeriodEnum*	A time period, e.g. a day, week, month or year of the stream. If the periodMultiplier value is 0 (zero) then period must contain the value D (day).
dayType	Optional *DayTypeEnum*	In the case of an offset specified as a number of days, this element defines whether consideration is given as to whether a day is a good business day or not. If a day type of business days is specified then non-business days are ignored when calculating the offset. The financial business centers to use for determination of business days are implied by the context in which this element is used. This element must only be included when the offset is specified as a number of days. If the offset is zero days then the dayType element should not be included.

instance Eq *DateOffset*

instance Show *DateOffset*

data *DateRelativeToEnum*

The specification of whether payments/resets occur relative to the first or last day of a calculation period.

CalculationPeriodStartDate

Payments/Resets will occur relative to the first day of each calculation period.

CalculationPeriodEndDate

Payments/Resets will occur relative to the last day of each calculation period.

instance Eq *DateRelativeToEnum*

instance Show *DateRelativeToEnum*

data *DayTypeEnum*

A day type classification used in counting the number of days between two dates.

Business

When calculating the number of days between two dates the count includes only business days.

Calendar

When calculating the number of days between two dates the count includes all calendar days.

CommodityBusiness

When calculating the number of days between two dates the count includes only commodity business days.

CurrencyBusiness

When calculating the number of days between two dates the count includes only currency business days.

ExchangeBusiness

When calculating the number of days between two dates the count includes only stock exchange business days.

ScheduledTradingDay

When calculating the number of days between two dates the count includes only scheduled trading days.

instance Eq *DayTypeEnum*

instance Show *DayTypeEnum*

data *FixedRateSchedule*

Specify the fixed rate

FixedRateSchedule

Field	Type	Description
initialValue	Decimal	The initial rate or amount, as the case may be. An initial rate of 5% would be represented as 0.05.
step	[*Step*]	The schedule of step date and value pairs. On each step date the associated step value becomes effective. A list of steps may be ordered in the document by ascending step date. An FpML document containing an unordered list of steps is still regarded as a conformant document. type_ : Optional SpreadScheduleType

instance Eq *FixedRateSchedule*

instance Show *FixedRateSchedule*

data *FixingDates*

Specifies the fixing date relative to the reset date in terms of a business days offset and an associated set of financial business centers.

FixingDates

Field	Type	Description
periodMultiplier	Int	
period	*PeriodEnum*	
dayType	Optional *DayTypeEnum*	
businessDayConvention	*BusinessDayConventionEnum*	
businessCenters	[Text]	

instance Eq *FixingDates*

instance Show *FixingDates*

data *FloatingRateCalculation*

A type defining the floating rate and definitions relating to the calculation of floating rate amounts.

FloatingRateCalculation

Field	Type	Description
floatingRateIndex	Text	
indexTenor	Optional *Period*	The ISDA Designated Maturity, i.e., the tenor of the floating rate. floatingRateMultiplierSchedule : Optional Schedule ^ A rate multiplier or multiplier schedule to apply to the floating rate. A multiplier schedule is expressed as explicit multipliers and dates. In the case of a schedule, the step dates may be subject to adjustment in accordance with any adjustments specified in the calculationPeriodDatesAdjustments. The multiplier can be a positive or negative decimal. This element should only be included if the multiplier is not equal to 1 (one) for the term of the stream.
spreadSchedule	[*SpreadSchedule*]	The ISDA Spread or a Spread schedule expressed as explicit spreads and dates. In the case of a schedule, the step dates may be subject to adjustment in accordance with any adjustments specified in calculationPeriodDatesAdjustments. The spread is a per annum rate, expressed as a decimal. For purposes of determining a calculation period amount, if positive the spread will be added to the floating rate and if negative the spread will be subtracted from the floating rate. A positive 10 basis point (0.1%) spread would be represented as 0.001. rateTreatment : Optional RateTreatmentEnum ^ The specification of any rate conversion which needs to be applied to the observed rate before being used in any calculations. The two common conversions are for securities quoted on a bank discount basis which will need to be converted to either a Money Market Yield or Bond Equivalent Yield. See the Annex to the 2000 ISDA Definitions, Section 7.3. Certain General Definitions Relating to Floating Rate Options, paragraphs (g) and (h) for definitions of these terms. capRateSchedule : [StrikeSchedule] ^ The cap rate or cap rate schedule, if any, which applies to the floating rate. The cap rate (strike) is only required where the floating rate on a swap stream is capped at a certain level. A cap rate schedule is expressed as explicit cap rates and dates and the step dates may be subject to adjustment in accordance with any adjustments specified in calculationPeriodDatesAdjustments. The cap rate is

instance Eq *FloatingRateCalculation*

instance Show *FloatingRateCalculation*

data *FxLinkedNotionalSchedule*

The notional amount or notional amount schedule (FX linked).

FxLinkedNotionalSchedule

Field	Type	Description
constantNotionalScheduleReference	Text	
initialValue	Optional Decimal	
varyingNotionalCurrency	Text	
varyingNotionalFixingDates	*FixingDates*	
fxSpotRateSource	*FxSpotRateSource*	

instance Eq *FxLinkedNotionalSchedule*

instance Show *FxLinkedNotionalSchedule*

data *FxSpotRateSource*

A type defining the rate source and fixing time for an fx rate.

FxSpotRateSource

Field	Type	Description
primaryRateSource	*InformationSource*	The primary source for where the rate observation will occur. Will typically be either a page or a reference bank published rate. secondaryRateSource : Optional InformationSource ^ An alternative, or secondary, source for where the rate observation will occur. Will typically be either a page or a reference bank published rate.
fixingTime	Optional *BusinessCenterTime*	The time at which the spot currency exchange rate will be observed. It is specified as a time in a business day calendar location, e.g., 11:00am London time.

instance Eq *FxSpotRateSource*

instance Show *FxSpotRateSource*

type ***HourMinuteTime*** = Text

A type defining a time specified in hh:mm:ss format where the second component must be '00', e.g., 11am would be represented as 11:00:00.

data *InformationSource*

A type defining the source for a piece of information (e.g. a rate refix or an fx fixing).

InformationSource

Field	Type	Description
rateSource	Text	An information source for obtaining a market rate. For example, Bloomberg, Reuters, Telerate etc. rateSourcePage : Optional RateSourcePage
rateSourcePage	Text	A specific page for the rate source for obtaining a market rate. rateSourcePage-Heading : Optional String ^ The heading for the rate source on a given rate source page.

instance Eq *InformationSource*

instance Show *InformationSource*

data *NotionalSchedule*

The notional amount or notional amount schedule.

NotionalSchedule

Field	Type	Description
id	Text	
notionalStep-Schedule	*Notional-StepSched-ule*	

instance Eq *NotionalSchedule*

instance Show *NotionalSchedule*

data *NotionalScheduleValue*

Specifies how the notional schedule is defined: either regular or fx linked.

NotionalSchedule_Regular NotionalSchedule

Regular notional schedule.

NotionalSchedule_FxLinked FxLinkedNotionalSchedule

FX linked notional schedule.

instance Eq *NotionalScheduleValue*

instance Show *NotionalScheduleValue*

data *NotionalStepSchedule*

The notional amount or notional amount schedule expressed as explicit outstanding notional amounts and dates.

NotionalStepSchedule

Field	Type	Description
initialValue	Decimal	
step	[*Step*]	
currency	Text	

instance Eq *NotionalStepSchedule*

instance Show *NotionalStepSchedule*

data *PaymentDates*

The payment dates schedule.

PaymentDates

Field	Type	Description
calculationPeriodDatesReference	Text	
paymentFrequency	*PaymentFrequency*	
firstPaymentDate	Optional Date	
lastRegularPaymentDate	Optional Date	
payRelativeTo	*DateRelativeToEnum*	
paymentDaysOffset	Optional *DateOffset*	
paymentDatesAdjustments	*BusinessDayAdjustments*	

instance Eq *PaymentDates*

instance Show *PaymentDates*

data *PaymentFrequency*

The frequency at which regular payment dates occur. If the payment frequency is equal to the frequency defined in the calculation period dates component then one calculation period contributes to each payment amount. If the payment frequency is less frequent than the frequency defined in the calculation period dates component then more than one calculation period will contribute to the payment amount. A payment frequency more frequent than the calculation period frequency or one that is not a multiple of the calculation period frequency is invalid. If the payment frequency is of value T (term), the period is defined by the swap\swapStream\calculationPerioDates\effectiveDate and the swap\swapStream\calculationPerioDates\terminationDate.

PaymentFrequency

Field	Type	Description
periodMultiplier	Int	
period	*PeriodExtendedEnum*	

instance Eq *PaymentFrequency*

instance Show *PaymentFrequency*

data *PeriodExtendedEnum*

The period of a schedule, for example the calculation schedule.

Regular PeriodEnum

T

instance Eq *PeriodExtendedEnum*

instance Show *PeriodExtendedEnum*

data *PrincipalExchanges*

A type defining which principal exchanges occur for the stream.

PrincipalExchanges

Field	Type	Description
initialExchange	Bool	A true/false flag to indicate whether there is an initial exchange of principal on the effective date.
finalExchange	Bool	A true/false flag to indicate whether there is a final exchange of principal on the termination date.
intermediateExchange	Bool	A true/false flag to indicate whether there are intermediate or interim exchanges of principal during the term of the swap.

instance Eq *PrincipalExchanges*

instance Show *PrincipalExchanges*

data *RateTypeValue*

Specifies whether the swapStream has a fixed or a floating rate.

RateType_Fixed FixedRateSchedule

Fixed rate.

RateType_Floating FloatingRateCalculation

Floating rate.

instance Eq *RateTypeValue*

instance Show *RateTypeValue*

data *ResetDates*

The reset dates schedule. This only applies for a floating rate stream.

ResetDates

Field	Type	Description
calculationPeriod-DatesReference	Text	
resetRelativeTo	*DateRela-tiveToEnum*	
fixingDates	*FixingDates*	
resetFrequency	*ResetFre-quency*	
resetDatesAdjust-ments	*ResetDate-sAdjust-ments*	

instance Eq *ResetDates*

instance Show *ResetDates*

data *ResetDatesAdjustments*

The business day convention to apply to each reset date if it would otherwise fall on a day that is not a business day in the specified financial business centers.

ResetDatesAdjustments

Field	Type	Description
businessDayCon-vention	*Business-DayConven-tionEnum*	
businessCenters	[Text]	

instance Eq *ResetDatesAdjustments*

instance Show *ResetDatesAdjustments*

data *ResetFrequency*

The frequency at which reset dates occur.

ResetFrequency

Field	Type	Description
periodMultiplier	Int	
period	*PeriodEx-tendedEnum*	

instance Eq *ResetFrequency*

instance Show *ResetFrequency*

data *Rounding*

A type defining a rounding direction and precision to be used in the rounding of a rate.

Rounding

Field	Type	Description
roundingDirection	*RoundingDirectionEnum*	Specifies the rounding direction.
precision	Int	Specifies the rounding precision in terms of a number of decimal places. Note how a percentage rate rounding of 5 decimal places is expressed as a rounding precision of 7 in the FpML document since the percentage is expressed as a decimal, e.g. 9.876543% (or 0.09876543) being rounded to the nearest 5 decimal places is 9.87654% (or 0.0987654).

instance Eq *Rounding*

instance Show *Rounding*

data *RoundingDirectionEnum*

The method of rounding a fractional number.

Up

A fractional number will be rounded up to the specified number of decimal places (the precision). For example, 5.21 and 5.25 rounded up to 1 decimal place are 5.3 and 5.3 respectively.

Down

A fractional number will be rounded down to the specified number of decimal places (the precision). For example, 5.29 and 5.25 rounded down to 1 decimal place are 5.2 and 5.2 respectively.

Nearest

A fractional number will be rounded either up or down to the specified number of decimal places (the precision) depending on its value. For example, 5.24 would be rounded down to 5.2 and 5.25 would be rounded up to 5.3 if a precision of 1 decimal place were specified.

instance Eq *RoundingDirectionEnum*

instance Show *RoundingDirectionEnum*

data *SpreadSchedule*

Adds an optional spread type element to the Schedule to identify a long or short spread value.

SpreadSchedule

Field	Type	Description
initialValue	Decimal	The initial rate or amount, as the case may be. An initial rate of 5% would be represented as 0.05. step : [Step] ^ The schedule of step date and value pairs. On each step date the associated step value becomes effective. A list of steps may be ordered in the document by ascending step date. An FpML document containing an unordered list of steps is still regarded as a conformant document. type_ : Optional SpreadScheduleType

instance Eq *SpreadSchedule*

instance Show *SpreadSchedule*

data *Step*

The schedule of step date and non-negative value pairs. On each step date the associated step value becomes effective. A list of steps may be ordered in the document by ascending step date. An FpML document containing an unordered list of steps is still regarded as a conformant document.

Step

Field	Type	Description
stepDate	Date	
stepValue	Decimal	

instance Eq *Step*

instance Show *Step*

data *StubCalculationPeriodAmount*

The stub calculation period amount parameters. This element must only be included if there is an initial or final stub calculation period. Even then, it must only be included if either the stub references a different floating rate tenor to the regular calculation periods, or if the stub is calculated as a linear interpolation of two different floating rate tenors, or if a specific stub rate or stub amount has been negotiated.

StubCalculationPeriodAmount

Field	Type	Description
calculationPeriod-DatesReference	Text	
initialStub	Optional *StubValue*	
finalStub	Optional *StubValue*	

instance Eq *StubCalculationPeriodAmount*

instance Show *StubCalculationPeriodAmount*

data *StubFloatingRate*

The rates to be applied to the initial or final stub may be the linear interpolation of two different rates.

StubFloatingRate

Field	Type	Description
floatingRateIndex	Text	
indexTenor	Optional *Period*	

instance Eq *StubFloatingRate*

instance Show *StubFloatingRate*

data *StubValue*

Specifies how the stub amount is calculated. A single floating rate tenor different to that used for the regular part of the calculation periods schedule may be specified, or two floating tenors may be specified. If two floating rate tenors are specified then Linear Interpolation (in accordance with the 2000 ISDA Definitions, Section 8.3. Interpolation) is assumed to apply. Alternatively, an actual known stub rate or stub amount may be specified.

StubValue_FloatingRate [*StubFloatingRate*]

The rates to be applied to the initial or final stub may be the linear interpolation of two different rates. While the majority of the time, the rate indices will be the same as that specified in the stream and only the tenor itself will be different, it is possible to specift two different rates. For example, a 2 month stub period may use the linear interpolation of a 1 month and 3 month rate. The different rates would be specified in this component. Note that a maximum of two rates can be specified. If a stub period uses the same floating rate index, including tenor, as the regular calculation periods then this should not be specified again within this component, i.e., the stub calculation period amount component may not need to be specified even if there is an initial or final stub period. If a stub period uses a different floating rate index compared to the regular calculation periods then this should be specified within this component. If specified here, they are likely to have id attributes, allowing them to be referenced from within the cashflows component.

StubValue_StubRate Decimal

An actual rate to apply for the initial or final stub period may have been agreed between the principal parties (in a similar way to how an initial rate may have been agreed for the first regular period). If an actual stub rate has been agreed then it would be included in this component. It will be a per annum rate, expressed as a decimal. A stub rate of 5% would be represented as 0.05. | StubValue_StubAmount Money ^ An actual amount to apply for the initial or final stub period may have been agreed between the two parties. If an actual stub amount has been agreed then it would be included in this component.

instance Eq *StubValue*

instance Show *StubValue*

data *SwapStream*

The swap streams, describing each leg of the swap.

SwapStream

Field	Type	Description
payerPartyReference	Text	
receiverPartyReference	Text	
calculationPeriodDates	*CalculationPeriodDates*	
paymentDates	*PaymentDates*	
resetDates	Optional *ResetDates*	
calculationPeriodAmount	*CalculationPeriodAmount*	
stubCalculationPeriodAmount	Optional *StubCalculationPeriodAmount*	
principalExchanges	Optional *PrincipalExchanges*	

instance Eq *SwapStream*

instance Show *SwapStream*

1.22.3.132 Daml.Finance.Interface.Instrument.Swap.Fpml.Instrument

Interfaces

interface *Instrument*

Instrument interface representing a swap specified as FpML swapStreams.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *V*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type *I* = *Instrument*
Type synonym for `Instrument`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Instrument V*

data *View*

View of `Instrument`.

View

Field	Type	Description
fpml	*Fpml*	Attributes of a swap specified as FpML swapStreams.

instance Eq *View*

instance Show *View*

1.22.3.133 Daml.Finance.Interface.Instrument.Swap.Fpml.Types

Data Types

data *Fpml*

Fpml

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.
description	Text	The description of the swap.
swapStreams	[*SwapStream*]	Each element describes a stream of swap payments, for example a regular fixed or floating rate.
issuerPartyRef	Text	Used to the identify which counterparty is the issuer in the swapStream.
calendarDataProvider	Party	The reference data provider to use for the holiday calendar.
currencies	[*InstrumentKey*]	The currencies of the different swap legs, one for each swapStream. For example, if one leg pays in USD this should be a USD cash instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

instance Eq *Fpml*

instance Show *Fpml*

1.22.3.134 Daml.Finance.Interface.Instrument.Swap.InterestRate.Factory

Interfaces

interface *Factory*

Factory interface to instantiate interest rate swaps.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new instrument.
Controller: (DA.Internal.Record.getField @"depository" (DA.Internal.Record.getField @"instrument" interestRate)), (DA.Internal.Record.getField @"issuer" (DA.Internal.Record.getField @"instrument" interestRate))
Returns: ContractId *I*

Field	Type	Description
interestRate	*InterestRate*	Attributes to create an interest rate swap.
observers	*PartiesMap*	The instrument's observers.

Choice *Remove*
Archive an instrument.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ()

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.
Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type ***F*** = *Factory*
Type synonym for `Factory`.

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View of `Factory`.

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.135 Daml.Finance.Interface.Instrument.Swap.InterestRate.Instrument

Interfaces

interface *Instrument*

Instrument interface representing an interest rate swap.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *V*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type ***I*** = *Instrument*
Type synonym for `Instrument`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Instrument V*

data *View*

View of `Instrument`.

View

Field	Type	Description
interestRate	*InterestRate*	Attributes of an interest rate swap.

instance Eq *View*

instance Show *View*

1.22.3.136 Daml.Finance.Interface.Instrument.Swap.InterestRate.Types

Data Types

data *InterestRate*

InterestRate

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.
description	Text	The description of the swap.
referenceRateId	Text	The floating rate reference ID. For example, in case of "3M Euribor vs 2.5% fix" this should be a valid reference to the "3M Euribor" reference rate.
ownerReceivesFix	Bool	Indicate whether a holding owner of this instrument receives the fix or the floating leg of the swap.
fixRate	Decimal	The interest rate of the fix leg. For example, in case of "3M Euribor vs 2.5% fix" this should be 0.025.
periodicSchedule	*PeriodicSchedule*	The schedule for the periodic swap payments.
holidayCalendarIds	[Text]	The identifiers of the holiday calendars to be used for the swap payment schedule.
calendarDataProvider	Party	The reference data provider to use for the holiday calendar.
dayCountConvention	*DayCountConventionEnum*	The day count convention used to calculate day count fractions. For example: Act360.
currency	*InstrumentKey*	The currency of the swap. For example, if the swap pays in USD this should be a USD cash instrument.
lastEventTimestamp	Time	(Market) time of the last recorded lifecycle event. If no event has occurred yet, the time of creation should be used.

instance Eq *InterestRate*

instance Show *InterestRate*

1.22.3.137 Daml.Finance.Interface.Instrument.Token.Factory

Interfaces

interface *Factory*

Factory interface to instantiate simple tokens.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new token.
Controller: (DA.Internal.Record.getField @"depository" (DA.Internal.Record.getField @"instrument" token)), (DA.Internal.Record.getField @"issuer" (DA.Internal.Record.getField @"instrument" token))
Returns: ContractId *I*

Field	Type	Description
token	*Token*	Attributes to create a token.
observers	*PartiesMap*	The instrument's observers.

Choice *Remove*
Archive a token.
Controller: (DA.Internal.Record.getField @"depository" instrument), (DA.Internal.Record.getField @"issuer" instrument)
Returns: ()

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.
Method remove : *Remove* -> Update ()
Implementation of `Remove` choice.

Data Types

type *F* = *Factory*
Type synonym for `Factory`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

remove : *Factory* -> *Remove* -> Update ()

1.22.3.138 Daml.Finance.Interface.Instrument.Token.Instrument

Interfaces

interface *Instrument*

Interface for a Token, an instrument whose economic terms on the ledger are represented by an `id` and a textual `description`.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *V*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type ***I*** = *Instrument*
Type synonym for `Instrument`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Instrument V*

data *View*

View of `Instrument`.

View

Field	Type	Description
token	*Token*	Attributes of a Token Instrument.

instance Eq *View*

instance Show *View*

1.22.3.139 Daml.Finance.Interface.Instrument.Token.Types

Data Types

data *Token*

Describes the attributes of a Token Instrument.

Token

Field	Type	Description
instrument	*InstrumentKey*	The instrument's key.
description	Text	A description of the instrument.
validAsOf	Time	Timestamp as of which the instrument is valid.

instance Eq *Token*

instance Show *Token*

1.22.3.140 Daml.Finance.Interface.Lifecycle.Effect

Interfaces

interface *Effect*

Interface for contracts exposing effects of lifecycling processes.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Calculate*
Given a holding, it calculates the instrument quantities to settle.
Controller: actor
Returns: *CalculationResult*

Field	Type	Description
actor	Party	The party calculating the quantities to settle.
holdingCid	ContractId *I*	The holding being targeted.

Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Choice *SetProviders*
Set the provider of the effect. The provider has visibility on all sub-transactions triggered by `Claim`\ing an effect.
Controller: (DA.Internal.Record.getField @"providers" (view this)), newProviders
Returns: ContractId *Effect*

Field	Type	Description
newProviders	*Parties*	The new provider.

Method calculate : *Calculate* -> ContractId *Effect* -> Update *CalculationResult*
Implementation of the `Calculate` choice.
Method setProviders : *SetProviders* -> Update (ContractId *Effect*)
Implementation of the `SetProviders` choice.

Data Types

data *CalculationResult*

Data type encapsulating the effect's calculation result.

CalculationResult

Field	Type	Description
consumed	[*InstrumentQuantity*]	Consumed quantities.
produced	[*InstrumentQuantity*]	Produced quantities.

instance Eq *CalculationResult*

instance Show *CalculationResult*

instance HasMethod *Effect* "calculate" (*Calculate* -> ContractId *Effect* -> Update *CalculationResult*)

type *I* = *Effect*
Type synonym for `Effect`.
instance HasMethod *Election* "apply" (ContractId *Election* -> *Apply* -> Update (Optional *InstrumentKey*, [ContractId *I*]))

instance HasMethod *Exercisable* "applyElection" (*ApplyElection* -> Update (Optional *InstrumentKey*, [ContractId *I*]))
instance HasMethod *Lifecycle* "evolve" (*Evolve* -> Update (Optional *InstrumentKey*, [ContractId *I*]))

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Effect V*

data *View*

View for `Effect`.

View

Field	Type	Description
providers	*Parties*	The parties providing the claim processing.
targetInstrument	*InstrumentKey*	The target instrument. A holding on this instrument is required to claim the effect. For example, in the case of a swap instrument, this would be the original instrument version before lifecycling, that contains the current swap payment.
producedInstrument	Optional *InstrumentKey*	The produced instrument, if it exists. For example, in the case of a swap instrument, this would be the new instrument version after lifecycling, that does not contain the current swap payment. If there are no more claims remaining after the current lifecycling, this would be None.
id	*Id*	A textual identifier.
description	Text	A human readable description of the Effect.
settlementTime	Optional Time	The effect's settlement time (if any).
otherConsumed	[*InstrumentQuantity*]	Consumed quantities (in addition to the target instrument). For example, in the case of a fix vs floating rate swap, this could be a 2.5% fix payment.
otherProduced	[*InstrumentQuantity*]	Produced quantities (in additon to the produced instrument). For example, in the case of a fix vs floating rate swap, this could be a 3M Euribor floating payment.

instance Eq *View*

instance Show *View*

Functions

setProviders : *Effect* -> *SetProviders* -> Update (ContractId *Effect*)

calculate : *Effect* -> *Calculate* -> ContractId *Effect* -> Update *CalculationResult*

1.22.3.141 Daml.Finance.Interface.Lifecycle.Election

Interfaces

interface *Election*

Interface implemented by templates that represents a claim-based election. This interface requires the `Event` interface implementation.

viewtype *V*

Choice *Apply*
Applies the election to the instrument, returning the new instrument as well as the corresponding effects. The election is archived as part of this choice.
Controller: (DA.Internal.Record.getField @"provider" (view this))
Returns: (Optional *InstrumentKey*, [ContractId *I*])

Field	Type	Description
observableCids	[ContractId *I*]	Set of observables.
exercisableCid	ContractId *Exercisable*	The contract that is used to apply an election to the instrument.

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Method apply : ContractId *Election* -> *Apply* -> Update (Optional *InstrumentKey*, [ContractId *I*])
Implementation of the `Apply` choice.

interface *Exercisable*

Interface implemented by instruments that admit (claim-based) elections.

viewtype *ExercisableView*

Choice *ApplyElection*
Applies an election to the instrument.
Controller: (DA.Internal.Record.getField @"lifecycler" (view this))
Returns: (Optional *InstrumentKey*, [ContractId *I*])

Field	Type	Description
electionCid	ContractId *Election*	The election.
observableCids	[ContractId *I*]	Set of observables.

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Exercisable_GetView*
Retrieves the interface view.
Controller: viewer
Returns: *ExercisableView*

Field	Type	Description
viewer	Party	The party retrieving the view.

Method applyElection : *ApplyElection* -> Update (Optional *InstrumentKey*, [ContractId *I*])
Implementation of the `ApplyElection` choice.

Data Types

data *ExercisableView*

View for `Exercisable`.

ExercisableView

Field	Type	Description
lifecycler	Party	Party processing the election.

instance Eq *ExercisableView*

instance Show *ExercisableView*

instance HasFromAnyView *Exercisable ExercisableView*

type *I* = *Election*
Type synonym for `Election`.
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))
instance HasMethod *Factory* "create'" (*Create* -> Update (ContractId *I*))

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Election V*

data *View*

View for `Election`.

View

Field	Type	Description
id	*Id*	The identifier for an election.
description	Text	A description of the instrument.
claim	Text	The tag corresponding to the elected sub-tree.
elector	Party	Parties on behalf of which the election is made.
counterparty	Party	Faces the `elector` in the `Holding`.
electorIsOwner	Bool	`True` if election is on behalf of the owner of the holding, `False` otherwise.
observers	*PartiesMap*	Observers of the election.
amount	Decimal	Number of instrument units to which the election applies.
provider	Party	Party that is authorized to process the election and generate the new instrument version and effects.
instrument	*InstrumentKey*	The instrument to which the election applies.

instance Eq *View*

instance Show *View*

Functions

apply : *Election* -> ContractId *Election* -> *Apply* -> Update (Optional *InstrumentKey*, [ContractId *I*])

getElectionTime : *Election* -> Time

Retrieves the election's time.

applyElection : *Exercisable* -> *ApplyElection* -> Update (Optional *InstrumentKey*, [ContractId *I*])

1.22.3.142 Daml.Finance.Interface.Lifecycle.Election.Factory

Interfaces

interface *Factory*

Factory interface to instantiate elections on generic instruments.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Create*
Create a new Election.
Controller: actors
Returns: ContractId *I*

Field	Type	Description
actors	*Parties*	Parties calling the `Create` choice.
elector	Party	Parties on behalf of which the election is made.
counterparty	Party	Faces the `elector` in the `Holding`.
provider	Party	Party that signs the election (together with the elector).
id	*Id*	The identifier for an election.
description	Text	A description of the instrument.
claim	Text	The tag corresponding to the elected sub-tree.
electorIsOwner	Bool	`True` if election is on behalf of the owner of the holding, `False` otherwise.
electionTime	Time	Time at which the election is put forward.
observers	*PartiesMap*	Observers of the election.
amount	Decimal	Number of instrument units to which the election applies.
instrument	*InstrumentKey*	The instrument to which the election applies.

Method create' : *Create* -> Update (ContractId *I*)
Implementation of `Create` choice.

Data Types

type ***F*** = *Factory*
Type synonym for `Factory`.

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View

Field	Type	Description
provider	Party	The provider of the `Factory`.

instance Eq *View*

instance Show *View*

Functions

create' : *Factory* -> *Create* -> Update (ContractId *I*)

1.22.3.143 Daml.Finance.Interface.Lifecycle.Event

Interfaces

interface *Event*

A lifecycle event. These events are ordered based on the corresponding event time.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type ***I*** = *Event*
Type synonym for `Event`.
instance HasMethod *Instrument* "declareDistribution" (*DeclareDistribution* -> Update (ContractId *I*))
instance HasMethod *Instrument* "declareReplacement" (*DeclareReplacement* -> Update (ContractId *I*))
instance HasMethod *Instrument* "declareStockSplit" (*DeclareStockSplit* -> Update (ContractId *I*))

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Event V*

data *View*

View for `Event`.

View

Field	Type	Description
providers	*Parties*	Providers of the event.
id	*Id*	Identifier for the event.
description	Text	A human readable description of the event.
eventTime	Time	The time of the event. This allows ordering of events.

instance Eq *View*

instance Show *View*

Functions

getEventTime : *Event* -> Time
Given an event, retrieves the event time.

1.22.3.144 Daml.Finance.Interface.Lifecycle.Event.Distribution

Interfaces

interface *Event*

Event interface for the distribution of units of an instrument for each unit of a target instrument (e.g. share or cash dividends).

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the event view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type ***I*** = *Event*
Type synonym for `Event`.

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Event V*

data *View*

View for `Event`.

View

Field	Type	Description
effectiveTime	Time	Time on which the distribution is effectuated.
targetInstrument	*InstrumentKey*	Instrument the distribution event applies to.
newInstrument	*InstrumentKey*	Instrument after the distribution has been claimed.
perUnitDistribution	[*InstrumentQuantity*]	Distributed quantities per unit held.

instance Eq *View*

instance Show *View*

1.22.3.145 Daml.Finance.Interface.Lifecycle.Event.Replacement

Interfaces

interface *Event*

Event interface for the replacement of units of an instrument with a basket of other instruments (e.g. stock merger).

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the event view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Data Types

type *I* = *Event*
Type synonym for `Event`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Event V*

data *View*

View for `Event`.

View

Field	Type	Description
effectiveTime	Time	Time on which the replacement is effectuated.
targetInstrument	*InstrumentKey*	Instrument the replacement event applies to.
perUnitReplacement	[*InstrumentQuantity*]	Instrument quantities the target instrument is replaced with.

instance Eq *View*

instance Show *View*

1.22.3.146 Daml.Finance.Interface.Lifecycle.Event.Time

Interfaces

interface *Event*

Event interface for events that signal the passing of (business) time.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the event view. The event's time can be retrieved from the generic `Event` interface.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Method advance : ContractId *Time* -> *Advance* -> Update (ContractId *Time*, ContractId *Event*)
Implementation of the `Advance` choice.
Method rewind : ContractId *Time* -> *Rewind* -> Update (ContractId *Time*, ContractId *Event*)
Implementation of the `Rewind` choice.

Data Types

type *I* = *Event*
Type synonym for `Event`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Event V*

data *View*

View for `Event`.

View

(no fields)

instance Eq *View*

instance Show *View*

1.22.3.147 Daml.Finance.Interface.Lifecycle.Observable.NumericObservable

This module defines an interface for a `NumericObservable`, which is used to inspect time-dependent numerical values.

Interfaces

interface *NumericObservable*

An interface to inspect some (time-dependent) numerical values (e.g. a stock price or an interest rate fixing) required when processing a lifecycle rule.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Choice *Observe*
Observe the `Observable`.
Controller: actors
Returns: Decimal

Field	Type	Description
actors	*Parties*	Parties calling this 'Observe' choice.
t	Time	Time at which the value is observed.

Method observe : Time -> Update Decimal
Implementation of the `Observe` choice.

Data Types

type *I* = *NumericObservable*
Type synonym for `Observable`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *NumericObservable V*

data *View*

View for `Observable`.

View

Field	Type	Description
provider	Party	Party providing the observations.
id	*Id*	Textual reference to the observable.

instance Eq *View*

instance Show *View*

Functions

observe : *NumericObservable* -> Time -> Update Decimal

1.22.3.148 Daml.Finance.Interface.Lifecycle.Observable.TimeObservable

This module defines an interface for a `TimeObservable`, which is implemented by templates exposing time information.

Interfaces

interface *TimeObservable*

An interface to inspect a time value.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetTime*
Retrieves the current time.
Controller: actors

Returns: Time

Field	Type	Description
actors	*Parties*	The party retrieving the current time.

Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Method getTime : Update Time
Implementation of the `GetTime` choice.

Data Types

type *I* = *TimeObservable*
Type synonym for `TimeObservable`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *TimeObservable V*

data *View*

View for `TimeObservable`.

View

Field	Type	Description
providers	*Parties*	Parties providing the observation.
id	*Id*	Textual reference to the observable.

instance Eq *View*

instance Show *View*

Functions

getTime : *TimeObservable* -> Update Time

1.22.3.149 Daml.Finance.Interface.Lifecycle.Rule.Claim

Interfaces

interface *Claim*

Interface for contracts that allow holders to claim an `Effect` and generate `SettlementInstruction`\s.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *ClaimEffect*
Claim an effect and generate corresponding settlement instructions.
Controller: claimer
Returns: *ClaimResult*

Field	Type	Description
claimer	Party	The party claiming the effect.
holdingCids	[ContractId *I*]	The holdings to process.
effectCid	ContractId *I*	The effect to process.
batchId	*Id*	Identifier used for the generated settlement batch.

Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Method claimEffect : *ClaimEffect* -> Update *ClaimResult*
Implementation of the `ClaimEffect` choice.

Data Types

data *ClaimResult*

Data type wrapping the results of `Claim``ing an ``Effect`.

ClaimResult

Field	Type	Description
batchCid	ContractId *I*	Batch used to batch-settle settlement instructions.
instructionCids	[ContractId *I*]	Settlement instructions to settle all effect consequences.

instance Eq *ClaimResult*

instance Show *ClaimResult*

instance HasMethod *Claim* "claimEffect" (*ClaimEffect* -> Update *ClaimResult*)

type *I* = *Claim*

Type synonym for `Claim`.

type *V* = *View*

Type synonym for `View`.

instance HasFromAnyView *Claim V*

data *View*

View for `Settlement`.

View

Field	Type	Description
providers	*Parties*	Providers of the claim rule. Together with the actors of the `ClaimEffect` choice the authorization requirements to upgrade the holdings being claimed have to be met.
claimers	*Parties*	Any of the parties can claim an effect.
settlers	*Parties*	Any of the parties can trigger settlement of the resulting batch.
routeProviderCid	ContractId *I*	RouteProvider contract used to discover settlement routes.
settlementFactoryCid	ContractId *F*	Settlement factory contract used to create a `Batch` of `Instruction`\s.

instance Eq *View*

instance Show *View*

Functions

claimEffect : *Claim* -> *ClaimEffect* -> Update *ClaimResult*

1.22.3.150 Daml.Finance.Interface.Lifecycle.Rule.Lifecycle

Interfaces

interface *Lifecycle*

Interface implemented by instruments that can be lifecycled (either by the instrument itself or by a separate rule contract).

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Evolve*
Process an event. It returns a tuple of the lifecycled instrument (or the original instrument when the former does not exist) and the effects.
Controller: (DA.Internal.Record.getField @"lifecycler" (view this))
Returns: (Optional *InstrumentKey*, [ContractId *I*])

Field	Type	Description
eventCid	ContractId *I*	The event.
instrument	*InstrumentKey*	The target instrument.
observableCids	[ContractId *I*]	Set of numerical time-dependent observables.

Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Method evolve : *Evolve* -> Update (Optional *InstrumentKey*, [ContractId *I*])
Implementation of the `Evolve` choice.

Data Types

type *I* = *Lifecycle*
Type synonym for `Lifecycle`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Lifecycle V*

data *View*

View for `Lifecycle`.

View

Field	Type	Description
id	*Id*	Identifier for the rule contract.
description	Text	Textual description.
lifecycler	Party	Party performing the lifecycling.

instance Eq *View*

instance Show *View*

Functions

evolve : *Lifecycle* -> *Evolve* -> Update (Optional *InstrumentKey*, [ContractId *I*])

1.22.3.151 Daml.Finance.Interface.Settlement.Batch

Interfaces

interface *Batch*

An interface for atomically settling a batch of `Instruction`\s. The corresponding Instructions are referenced by key.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Cancel*
Cancels the batch.
Controller: actors
Returns: [ContractId *I*]

Field	Type	Description
actors	*Parties*	The parties canceling the batch.

Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Choice *Settle*
Settles the batch.
Controller: actors
Returns: [ContractId *I*]

Field	Type	Description
actors	*Parties*	The parties settling the batch.

Method cancel : *Cancel* -> Update [ContractId *I*]
Implementation of the `Cancel` choice.
Method settle : *Settle* -> Update [ContractId *I*]
Implementation of the `Settle` choice.

Data Types

type *I* = *Batch*
Type synonym for `Batch`.
instance HasMethod *Factory* "instruct" (*Instruct* -> Update (ContractId *I*, [ContractId *I*]))

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Batch V*

data *View*

View for `Batch`.

View

Field	Type	Description
requestors	*Parties*	Parties requesting the settlement.
settlers	*Parties*	Parties that can trigger the final settlement.
id	*Id*	Batch identifier.
description	Text	Batch description.
contextId	Optional *Id*	Identifier to link a batch to a context (e.g. the `Effect` it originated from).
routedSteps	[*RoutedStep*]	Routed settlement steps.
settlementTime	Optional Time	Settlement time (if any).

instance Eq *View*

instance Show *View*

Functions

settle : *Batch* -> *Settle* -> Update [ContractId *I*]

cancel : *Batch* -> *Cancel* -> Update [ContractId *I*]

1.22.3.152 Daml.Finance.Interface.Settlement.Factory

Interfaces

interface *Factory*

An interface used to generate settlement instructions.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()

(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Choice *Instruct*
Generate settlement instructions, and a batch for settling them.
Controller: instructors
Returns: (ContractId *I*, [ContractId *I*])

Field	Type	Description
instructors	*Parties*	Parties requesting to instruct a settlement.
settlers	*Parties*	Any of the parties can trigger the final settlement.
id	*Id*	Batch identifier.
description	Text	Batch description.
contextId	Optional *Id*	Identifier to link a batch to a context (e.g. the `Effect` it originated from).
routedSteps	[*RoutedStep*]	Routed settlement steps to instruct.
settlementTime	Optional Time	Settlement time (if any).

Method instruct : *Instruct* -> Update (ContractId *I*, [ContractId *I*])
Implementation of the `Instruct` choice.

Data Types

type ***F*** = *Factory*
Type synonym for `Factory`.

type ***I*** = *Factory*
Type synonym for `Factory`.

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Factory V*

data *View*

View for `Factory`.

View

Field	Type	Description
provider	Party	Party providing the facility.
observers	*Parties*	Observers.

instance Eq *View*

instance Show *View*

Functions

instruct : *Factory* -> *Instruct* -> Update (ContractId *I*, [ContractId *I*])

1.22.3.153 Daml.Finance.Interface.Settlement.Instruction

Interfaces

interface *Instruction*

An interface for providing a single instruction to transfer an asset.

viewtype *V*

Choice *Allocate*
Allocates this instruction and optionally returns a previously allocated (mutated) asset.
Controller: actors
Returns: (ContractId *Instruction*, Optional (ContractId *I*))

Field	Type	Description
actors	*Parties*	The parties allocating the instruction.
allocation	*Allocation*	Allocation of an instruction.

Choice *Approve*
Approves this instruction.
Controller: actors
Returns: ContractId *Instruction*

Field	Type	Description
actors	*Parties*	The parties approving the instruction.
approval	*Approval*	Approval of an instruction.

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Cancel*
Cancels this instruction.
Controller: actors
Returns: Optional (ContractId *I*)

Field	Type	Description
actors	*Parties*	The parties canceling the instruction.

Choice *Execute*
Executes this instruction.
Controller: actors
Returns: Optional (ContractId *I*)

Field	Type	Description
actors	*Parties*	The parties executing the instruction.

Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Method allocate : *Allocate* -> Update (ContractId *Instruction*, Optional (ContractId *I*))
Implementation of the `Allocate` choice.
Method approve : *Approve* -> Update (ContractId *Instruction*)
Implementation of the `Approve` choice.
Method cancel : *Cancel* -> Update (Optional (ContractId *I*))
Implementation of the `Cancel` choice.
Method execute : *Execute* -> Update (Optional (ContractId *I*))
Implementation of the `Execute` choice.

Data Types

type ***I*** = *Instruction*
Type synonym for `Instruction`.
instance HasMethod *Factory* "instruct" (*Instruct* -> Update (ContractId *I*, [ContractId *I*]))

type ***V*** = *View*
Type synonym for `View`.
instance HasFromAnyView *Instruction V*

data *View*

View for `Instruction`.

View

Field	Type	Description
requestors	*Parties*	Parties that instructed settlement.
settlers	*Parties*	Parties that can execute the Instruction.
batchId	*Id*	Batch identifier.
id	*Id*	Instruction identifier.
routedStep	*RoutedStep*	Instruction details to execute.
settlementTime	Optional Time	Settlement time (if any).
allocation	*Allocation*	Allocation from the sender.
approval	*Approval*	Approval from the receiver.
signedSenders	*Parties*	Additional signatories, used to collect authorization (on sending side).
signedReceivers	*Parties*	Additional signatories, used to collect authorization (on receiving side).

instance Eq *View*

instance Show *View*

Functions

allocate : *Instruction* -> *Allocate* -> Update (ContractId *Instruction*, Optional (ContractId *I*))

approve : *Instruction* -> *Approve* -> Update (ContractId *Instruction*)

execute : *Instruction* -> *Execute* -> Update (Optional (ContractId *I*))

cancel : *Instruction* -> *Cancel* -> Update (Optional (ContractId *I*))

1.22.3.154 Daml.Finance.Interface.Settlement.RouteProvider

Interfaces

interface *RouteProvider*

An interface used to discover the settlement route for each `Step`, i.e., `[RoutedStep]`.

viewtype *V*

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *Discover*
Discover the settlement route for each `Step`.
Controller: discoverors
Returns: [*RoutedStep*]

Field	Type	Description
discoverors	*Parties*	Parties requesting to discover.
contextId	Optional *Id*	Context for the discovery.
steps	[*Step*]	Settlement steps to route.

Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Method discover : *Discover* -> Update [*RoutedStep*]
Implementation of the `Discover` choice.

Data Types

type *I* = *RouteProvider*
Type synonym for `RouteProvider`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *RouteProvider V*

data *View*

View for `RouteProvider`.

View

Field	Type	Description
provider	Party	Party providing the `RouteProvider` facility.
observers	*Parties*	Observers.

instance Eq *View*

instance Show *View*

Functions

discover : *RouteProvider* -> *Discover* -> Update [*RoutedStep*]

1.22.3.155 Daml.Finance.Interface.Settlement.Types

Data Types

data *Allocation*

Describes an allocation of an `Instruction`.

Unallocated

An unallocated instruction.

Pledge (ContractId *I*)

Settle the instruction with the pledged asset.

CreditReceiver

Settle the instruction by crediting the receiver account (where the sender is custodian).

SettleOffledger

Settle the instruction off-ledger.

PassThroughFrom (*AccountKey*, *InstructionKey*)

Settle the instruction with the holding coming from the specified instruction and account.

instance Eq *Allocation*

instance Show *Allocation*

data *Approval*

Describes an approval of an `Instruction`.

Unapproved

An unapproved instruction.

TakeDelivery AccountKey

Take delivery to the specified account.

DebitSender

Debit the sender account with the provided asset (where the receiver is custodian).

SettleOffledgerAcknowledge

Acknowledge settlement of the instruction off-ledger.

PassThroughTo (*AccountKey*, *InstructionKey*)

Take delivery to the specified account. The holding is then immediately allocated to the specified instruction.

instance Eq *Approval*

instance Show *Approval*

data *InstructionKey*

A unique key for Instructions.

InstructionKey

Field	Type	Description
requestors	*Parties*	Parties requesting settlement of the instruction.
batchId	*Id*	Id of the batch the instruction belongs to.
id	*Id*	A unique identifier for an instruction.

instance Eq *InstructionKey*

instance Ord *InstructionKey*

instance Show *InstructionKey*

instance HasExerciseByKey *Instruction InstructionKey* Archive ()

instance HasFetchByKey *Instruction InstructionKey*

instance HasFromAnyContractKey *Instruction InstructionKey*

instance HasKey *Instruction InstructionKey*

instance HasLookupByKey *Instruction InstructionKey*

instance HasMaintainer *Instruction InstructionKey*

instance HasToAnyContractKey *Instruction InstructionKey*

data *RoutedStep*

Describes a transfer of a position between two parties. The custodian at which the position is held is also specified.

RoutedStep

Field	Type	Description
sender	Party	Party transferring the asset.
receiver	Party	Party receiving the asset.
custodian	Party	The custodian at which the asset is held.
quantity	*InstrumentQuantity*	The instrument and amount to be transferred.

instance Eq *RoutedStep*

instance Ord *RoutedStep*

instance Show *RoutedStep*

instance HasMethod *RouteProvider* "discover" (*Discover* -> Update [*RoutedStep*])

data *Step*

Describes a transfer of a position between two parties.

Step

Field	Type	Description
sender	Party	Party transferring the asset.
receiver	Party	Party receiving the asset.
quantity	*InstrumentQuantity*	The instrument and amount to be transferred.

instance Eq *Step*

instance Ord *Step*

instance Show *Step*

1.22.3.156 Daml.Finance.Interface.Types.Common.Types

Data Types

data *AccountKey*

A unique key for Accounts.

AccountKey

Field	Type	Description
custodian	Party	Account provider.
owner	Party	Account owner.
id	*Id*	Unique identifier for an account.

instance Eq *AccountKey*

instance Ord *AccountKey*

instance Show *AccountKey*

instance HasMethod *Account* "getKey" *AccountKey*

instance HasExerciseByKey Reference *AccountKey* GetCid (ContractId *Account*)

instance HasExerciseByKey Reference *AccountKey* SetCid (ContractId Reference)

instance HasExerciseByKey Reference *AccountKey* SetObservers (ContractId Reference)

instance HasExerciseByKey Reference *AccountKey* Archive ()

instance HasFetchByKey Reference *AccountKey*

instance HasFromAnyContractKey Reference *AccountKey*

instance HasKey Reference *AccountKey*

instance HasLookupByKey Reference *AccountKey*

instance HasMaintainer Reference *AccountKey*

instance HasToAnyContractKey Reference *AccountKey*

data *Id*

Id Text

instance Eq *Id*

instance Ord *Id*

instance Show *Id*

instance HasExerciseByKey *LedgerTime* (*Parties*, *Id*) Archive ()

instance HasFetchByKey *LedgerTime* (*Parties*, *Id*)

instance HasFromAnyContractKey *LedgerTime* (*Parties*, *Id*)

instance HasKey *LedgerTime* (*Parties*, *Id*)

instance HasLookupByKey *LedgerTime* (*Parties*, *Id*)

instance HasMaintainer *LedgerTime* (*Parties*, *Id*)

instance HasToAnyContractKey *LedgerTime* (*Parties*, *Id*)

data *InstrumentKey*

A unique key for Instruments.

InstrumentKey

Field	Type	Description
depository	Party	Party providing depository services.
issuer	Party	Issuer of instrument.
id	*Id*	A unique identifier for an instrument.
version	Text	A textual instrument version.

instance Eq *InstrumentKey*

instance Ord *InstrumentKey*

instance Show *InstrumentKey*

instance HasMethod *Instrument* "getKey" *InstrumentKey*

instance HasMethod *Election* "apply" (ContractId *Election* -> *Apply* -> Update (Optional *InstrumentKey*, [ContractId *I*]))

instance HasMethod *Exercisable* "applyElection" (*ApplyElection* -> Update (Optional *InstrumentKey*, [ContractId *I*]))

instance HasMethod *Lifecycle* "evolve" (*Evolve* -> Update (Optional *InstrumentKey*, [ContractId *I*]))

instance HasExerciseByKey Reference *InstrumentKey* GetCid (ContractId *I*)

instance HasExerciseByKey Reference *InstrumentKey* SetCid (ContractId *R*)

instance HasExerciseByKey Reference *InstrumentKey* SetObservers (ContractId *R*)

instance HasExerciseByKey Reference *InstrumentKey* Archive ()

instance HasFetchByKey Reference *InstrumentKey*

instance HasFromAnyContractKey Reference *InstrumentKey*

instance HasKey Reference *InstrumentKey*

instance HasLookupByKey Reference *InstrumentKey*

instance HasMaintainer Reference *InstrumentKey*

instance HasToAnyContractKey Reference *InstrumentKey*

type ***InstrumentQuantity*** = *Quantity InstrumentKey* Decimal

type ***Parties*** = Set Party

A set of parties.

instance HasExerciseByKey *LedgerTime* (*Parties*, *Id*) Archive ()
instance HasFetchByKey *LedgerTime* (*Parties*, *Id*)
instance HasFromAnyContractKey *LedgerTime* (*Parties*, *Id*)
instance HasKey *LedgerTime* (*Parties*, *Id*)

instance HasLookupByKey *LedgerTime* (*Parties*, *Id*)
instance HasMaintainer *LedgerTime* (*Parties*, *Id*)
instance HasToAnyContractKey *LedgerTime* (*Parties*, *Id*)

type ***PartiesMap*** = Map Text *Parties*

Parties mapped by a specific key (or context). The textual key is the "context" which describes the value set of parties. This allows processes to add/remove parties for their specific purpose, without affecting others.

data *Quantity* u a

A dimensioned quantity.

Quantity

Field	Type	Description
unit	u	The quantity's unit.
amount	a	A numerical amount.

instance (Eq u, Eq a) => Eq (*Quantity* u a)

instance (Ord u, Ord a) => Ord (*Quantity* u a)

instance (Show u, Show a) => Show (*Quantity* u a)

1.22.3.157 Daml.Finance.Interface.Types.Date.Calendar

Data Types

data *BusinessDayAdjustment*

A data type to define how non-business days are adjusted.

BusinessDayAdjustment

Field	Type	Description
calendarIds	[Text]	A list of calendar ids to define holidays.
convention	*BusinessDayConventionEnum*	The business day convention used for the adjustment.

instance Eq *BusinessDayAdjustment*

instance Show *BusinessDayAdjustment*

data *BusinessDayConventionEnum*

An enum type to specify how a non-business day is adjusted.

Following

Adjust a non-business day to the next business day.

ModifiedFollowing

Adjust a non-business day to the next business day unless it is not in the same month. In this case use the previous business day.

ModifiedPreceding

Adjust a non-business day to the previous business day unless it is not in the same month. In this case use the next business day.

NoAdjustment

Non-business days are not adjusted.

Preceding

Adjust a non-business day to the previous business day.

instance Eq *BusinessDayConventionEnum*

instance Show *BusinessDayConventionEnum*

data *HolidayCalendarData*

Holiday Calendar Data used to define holidays (non-business days).

HolidayCalendarData

Field	Type	Description
id	Text	The id of the holiday calendar.
weekend	[DayOfWeek]	A list of week days defining the weekend.
holidays	[Date]	A list of dates defining holidays.

instance Eq *HolidayCalendarData*

instance Show *HolidayCalendarData*

instance HasExerciseByKey *HolidayCalendar HolidayCalendarKey GetCalendar HolidayCalendarData*

1.22.3.158 Daml.Finance.Interface.Types.Date.Classes

Typeclasses

class *HasUTCTimeConversion* a **where**

Types that can be converted to UTC time.

toUTCTime : a -> Time

instance *HasUTCTimeConversion DateClock*

instance *HasUTCTimeConversion Unit*

1.22.3.159 Daml.Finance.Interface.Types.Date.DayCount

Data Types

data *DayCountConventionEnum*

An enum type to specify a day count convention used to calculate day count fractions. For a detailed definition of each convention, we refer to the "Method of Interest Computation Indicator" definitions in the context of the ISO-20022 standard. Where useful, we provide disambiguation comments.

Act360

Actual 360. In CDM it is called *DayCountFractionEnum_ACT_360*. In ISO20022 it is called *A004*.

Act365Fixed

Actual 365 fixed. In CDM it is called *DayCountFractionEnum_ACT_365_FIXED*. In ISO20022 it is called *A005*.

Act365L

Actual 365L. In CDM it is called *DayCountFractionEnum_ACT_365L*. In ISO20022 it is called *A009*.

ActActAFB

Actual Actual AFB. In CDM it is called *DayCountFractionEnum_ACT_ACT_AFB*. In ISO20022 it is called *A010*.

ActActISDA

Actual Actual ISDA. In CDM it is called *DayCountFractionEnum_ACT_ACT_ISDA*. In ISO20022 it is called *A008*.

ActActICMA

Actual Actual ICMA. In CDM it is called *DayCountFractionEnum_ACT_ACT_ICMA* and *DayCountFractionEnum_ACT_ACT_ISMA* (they are identical: https://www.isda.org/2011/01/07/act-act-icma/). In ISO20022 it is called *A006*. Also called ISMA in the 1998 ISDA paper.

Basis1

1/1. In CDM it is called *DayCountFractionEnum__1_1*. Currently not included in ISO20022.

Basis30360

30/360. In CDM it is called *DayCountFractionEnum__30_360*. In ISO20022 it is called *A001*. Also called 30/360 ISDA or American Basic rule.

Basis30360ICMA

30/360 ICMA. In CDM it is called *DayCountFractionEnum__30E_360*. In ISO20022 it is called *A011*. Also called Basic Rule. This corresponds to "30E/360" of the 2006 ISDA definitions.

Basis30E360

30E/360. In CDM it is called *DayCountFractionEnum__30E_360_ISDA*. In ISO20022 it is called *A007*. Also called Eurobond basis. This corresponds to "30E360 (ISDA)" of the 2006 ISDA definitions.

Basis30E3360

30E3/360. Currently not included in CDM. In ISO20022 it is called *A013*. Also called Eurobond basis model 3.

instance Eq *DayCountConventionEnum*

instance Show *DayCountConventionEnum*

1.22.3.160 Daml.Finance.Interface.Types.Date.RollConvention

Data Types

data *Period*

A data type to define periods.

Period

Field	Type	Description
period	*PeriodEnum*	A period, e.g., a day, week, month or year.
periodMultiplier	Int	A period multiplier, e.g., 1, 2 or 3 etc.

instance Eq *Period*

instance Show *Period*

data *PeriodEnum*

An enum type to specify a period, e.g., day or week.

D

Day

M

Month

W

Week

Y

Year

instance Eq *PeriodEnum*

instance Show *PeriodEnum*

data *RollConventionEnum*

An enum type to specify how to roll dates.

EOM

Rolls on month end.

DOM Int

Rolls on the corresponding day of the month.

NoRollConvention

No roll convention is specified. This is for e.g. when date roll is not required (`D` or `W` tenors, single-period schedules).

instance Eq *RollConventionEnum*

instance Show *RollConventionEnum*

1.22.3.161 Daml.Finance.Interface.Types.Date.Schedule

Data Types

data *Frequency*

Frequency of a periodic schedule.

Frequency

Field	Type	Description
period	*Period*	The period (e.g., 1D, 3M, 1Y).
rollConvention	*RollConvention-tionEnum*	The roll convention.

instance Eq *Frequency*

instance Show *Frequency*

data *PeriodicSchedule*

A periodic schedule.

PeriodicSchedule

Field	Type	Description
effectiveDate	Date	Effective date, i.e., the (unadjusted) start date of the first period.
terminationDate	Date	Termination date, i.e., the (unadjusted) end date of the last period.
firstRegularPeriodStartDate	Optional Date	The (unadjusted) start date of the first regular period (optional).
lastRegularPeriodEndDate	Optional Date	The (unadjusted) end date of the last regular period (optional).
frequency	*ScheduleFrequency*	The frequency of the periodic schedule.
businessDayAdjustment	*BusinessDayAdjustment*	The business day adjustment to determine adjusted dates.
effectiveDateBusinessDayAdjustment	Optional *BusinessDayAdjustment*	The (optional) business day adjustment of the effective date
terminationDateBusinessDayAdjustment	Optional *BusinessDayAdjustment*	The (optional) business day adjustment of the termination date
stubPeriodType	Optional *StubPeriodTypeEnum*	An optional stub to define a stub implicitly and not via `firstRegularPeriodStartDate` or `lastRegularPeriodEndDate`.

instance Eq *PeriodicSchedule*

instance Show *PeriodicSchedule*

type ***Schedule*** = [*SchedulePeriod*]

A schedule defined by a list of periods.

data *ScheduleFrequency*

Frequency of a schedule. It can be specified as a regular frequency or as `SinglePeriod`.

Periodic Frequency

Periodic frequency (e.g. 1D, 3M, 1Y).

SinglePeriod

Used for schedules that have exactly one regular period covering their full term (from `effectiveDate` to `terminationDate`).

instance Eq *ScheduleFrequency*

instance Show *ScheduleFrequency*

data *SchedulePeriod*

A single period in a schedule.

SchedulePeriod

Field	Type	Description
adjustedEndDate	Date	Adjusted end date.
adjustedStartDate	Date	Adjusted start date.
unadjustedEnd-Date	Date	Unadjusted end date.
unadjustedStart-Date	Date	Unadjusted start date.
stubType	Optional *StubPeriod-TypeEnum*	Indicates whether this period is a stub (and if so, what type of stub it is)

instance Eq *SchedulePeriod*

instance Show *SchedulePeriod*

data *StubPeriodTypeEnum*

An enum type to specify a stub.

LongFinal

A long (more than one period) final stub.

LongInitial

A long (more than one period) initial stub.

ShortFinal

A short (less than one period) final stub.

ShortInitial

A short (less than one period) initial stub.

instance Eq *StubPeriodTypeEnum*

instance Show *StubPeriodTypeEnum*

1.22.3.162 Daml.Finance.Interface.Util.Common

Functions

verify : CanAssert m => Bool -> Text -> m ()
Verify is assertMsg with its arguments flipped.

fetchInterfaceByKey : (HasInterfaceTypeRep i, HasInterfaceTypeRep i2, HasFetchByKey t k, HasField "cid" t (ContractId i), HasFetch i2) => k -> Update i2
Fetch an interface by key.

qty : Decimal -> *InstrumentKey* -> *InstrumentQuantity*
Wraps an amount and an instrument key into an instrument quantity.

scale : Decimal -> *InstrumentQuantity* -> *InstrumentQuantity*
Scale `quantity` by the provided factor.

1.22.3.163 Daml.Finance.Interface.Util.Disclosure

Interfaces

interface *Disclosure*

An interface for managing the visibility of contracts for non-authorizing parties.

viewtype *V*

Choice *AddObservers*
Add a single new observer context to the existing observers.
Controller: disclosers
Returns: ContractId *Disclosure*

Field	Type	Description
disclosers	*Parties*	Party calling this choice.
observersToAdd	(Text, *Parties*)	Parties to add as observers to the contract and the corresponding observer context. If the observer context already exists, the new set of parties is added to the old one.

Choice Archive
Controller: Signatories of implementing template
Returns: ()
(no fields)
Choice *GetView*
Retrieves the interface view.
Controller: viewer
Returns: *View*

Field	Type	Description
viewer	Party	The party retrieving the view.

Choice *RemoveObservers*
Remove observers from a context. None is returned if no update is needed. Parties for a context can be removed if any of the disclosers are part of the observers to be removed or the disclosureControllers.
Controller: disclosers
Returns: Optional (ContractId *Disclosure*)

Field	Type	Description
disclosers	*Parties*	Parties calling this choice.
observersToRemove	(Text, *Parties*)	Parties to be removed from the contract observers and the corresponding observer context.

Choice *SetObservers*
Set the observers for a contract.
Controller: disclosers
Returns: ContractId *Disclosure*

Field	Type	Description
disclosers	*Parties*	Party calling this choice.
newObservers	*PartiesMap*	Observers to set for this contract. This overrides the existing observers. The parties are mapped by a specific key. The textual key is the "observation context" of the disclosure. This allows processes to add/remove parties for their specific purpose, without affecting others.

Method addObservers : *AddObservers* -> Update (ContractId *Disclosure*)
Implementation of the `AddObservers` choice.
Method removeObservers : ContractId *Disclosure* -> *RemoveObservers* -> Update (Optional (ContractId *Disclosure*))
Implementation of the `RemoveObservers` choice.
Method setObservers : *SetObservers* -> Update (ContractId *Disclosure*)
Implementation of the `SetObservers` choice.

Data Types

type *I* = *Disclosure*
Type synonym for `Disclosure`.

type *V* = *View*
Type synonym for `View`.
instance HasFromAnyView *Disclosure V*

data *View*

View for `Disclosure`.

View

Field	Type	Description
disclosureControllers	*Parties*	Disjunction choice controllers.
observers	*PartiesMap*	Observers with context. The parties are mapped by a specific key. The textual key is the "observation context" of the disclosure. This allows processes to add/remove parties for their specific purpose, without affecting others.

instance Eq *View*

instance Show *View*

Functions

setObservers : *Disclosure* -> *SetObservers* -> Update (ContractId *Disclosure*)

addObservers : *Disclosure* -> *AddObservers* -> Update (ContractId *Disclosure*)

removeObservers : *Disclosure* -> ContractId *Disclosure* -> *RemoveObservers* -> Update (Optional (ContractId *Disclosure*))

flattenObservers : *PartiesMap* -> *Parties*

Flattens observers which use the `PartiesMap` into a `Set Party` for usage in template definitions. For example:

```
observer $ flattenObservers observers
```

1.22.3.164 Daml.Finance.Lifecycle.Effect

Templates

template *Effect*

A contract encoding the consequences of a lifecycle event for one unit of the target instrument.

Signatory: providers

Field	Type	Description
providers	*Parties*	The effect provider.
id	*Id*	The effect's identifier.
description	Text	The effect's description.
targetInstrument	*InstrumentKey*	The target instrument. A holding on this instrument is required to claim the effect. For example, in the case of a swap instrument, this would be the original instrument version before lifecycling, that contains the current swap payment.
producedInstrument	Optional *InstrumentKey*	The produced instrument, if it exists. For example, in the case of a swap instrument, this would be the new instrument version after lifecycling, that does not contain the current swap payment. If there are no more claims remaining after the current lifecycling, this would be None.
otherConsumed	[*InstrumentQuantity*]	Consumed quantities (in addition to the target instrument). For example, in the case of a fix vs floating rate swap, this could be a 2.5% fix payment.
otherProduced	[*InstrumentQuantity*]	Produced quantities (in additon to the produced instrument). For example, in the case of a fix vs floating rate swap, this could be a 3M Euribor floating payment.
settlementTime	Optional Time	The effect's settlement time (if any).
observers	*PartiesMap*	Observers.

Choice Archive
Controller: providers
Returns: ()
(no fields)
interface instance *I* **for** *Effect*
interface instance *I* **for** *Effect*

Data Types

type *T* = *Effect*
Type synonym for `Effect`.

1.22.3.165 Daml.Finance.Lifecycle.Election

Templates

template *Election*

An election, such as the exercise of an option.

Signatory: elector, provider

Field	Type	Description
elector	Party	Entity making the election.
counterparty	Party	Faces the `elector` in the `Holding`.
provider	Party	The provider of the election is an entity that has the authority to process the election and create a new instrument version.
id	*Id*	Election identifier.
description	Text	A human readable description of the election.
instrument	*InstrumentKey*	The instrument to which the election applies.
amount	Decimal	Number of units of instrument to which the election applies.
claim	Text	The tag corresponding to the elected sub-tree.
electorIsOwner	Bool	`True` if the elector is the owner of a claim, `False` otherwise.
electionTime	Time	Time at which the election is put forward.
observers	*PartiesMap*	A set of observers.

Choice Archive
Controller: elector, provider
Returns: ()
(no fields)
interface instance *I* **for** *Election*
interface instance *I* **for** *Election*
interface instance *I* **for** *Election*

template *Factory*

Factory template to create an `Election`.

Signatory: provider

Field	Type	Description
provider	Party	The provider of the `Factory`.
observers	*PartiesMap*	A set of observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *F* **for** *Factory*
interface instance *I* **for** *Factory*

Data Types

type *T* = *Election*
Type synonym for `Election`.

1.22.3.166 Daml.Finance.Lifecycle.ElectionEffect

Templates

template *ElectionEffect*

A contract encoding the consequences of an election for one unit of the target instrument. It needs to be claimed with a holding of the right amount and is consumed after claiming.

Signatory: providers

Field	Type	Description
providers	*Parties*	The effect provider.
custodian	Party	The custodian of the holding put forward for election.
owner	Party	The owner of the holding put forward for election.
id	*Id*	The effect's identifier.
description	Text	The effect's description.
targetInstrument	*InstrumentKey*	The target instrument.
producedInstrument	Optional *InstrumentKey*	The produced instrument, when it exists.
amount	Decimal	The elected amount.
otherConsumed	[*InstrumentQuantity*]	Consumed quantities (not including the target instrument).
otherProduced	[*InstrumentQuantity*]	Produced quantities (not including the produced instrument).
settlementTime	Optional Time	The effect's settlement time (if any).
observers	*PartiesMap*	Observers.

Choice Archive
Controller: providers
Returns: ()
(no fields)
interface instance *I* **for** *ElectionEffect*
interface instance *I* **for** *ElectionEffect*

Data Types

type *T* = *ElectionEffect*
Type synonym for `ElectionEffect`.

1.22.3.167 Daml.Finance.Lifecycle.Event.Distribution

Templates

template *Event*

Event contract for the distribution of units of an instrument for each unit of a target instrument (e.g. share or cash dividends).

Signatory: providers

Field	Type	Description
providers	*Parties*	Providers of the distribution event.
id	*Id*	Event Identifier.
description	Text	Event description.
effectiveTime	Time	Time on which the distribution is effectuated.
targetInstrument	*InstrumentKey*	Instrument the distribution event applies to.
newInstrument	*InstrumentKey*	Instrument after the distribution has been claimed.
perUnitDistribution	[*InstrumentQuantity*]	Distributed quantities per unit held.
observers	*Parties*	Observers.

Choice Archive
Controller: providers
Returns: ()
(no fields)
interface instance *I* **for** *Event*
interface instance *I* **for** *Event*

Data Types

type *T* = *Event*
Type synonym for `Event`.

1.22.3.168 Daml.Finance.Lifecycle.Event.Replacement

Templates

template *Event*

Event contract for the replacement of units of an instrument with a basket of other instruments, e.g., a stock merger.

Signatory: providers

Field	Type	Description
providers	*Parties*	Providers of the distribution event.
id	*Id*	Event identifier.
description	Text	Event description.
effectiveTime	Time	Time on which the replacement is effectuated.
targetInstrument	*InstrumentKey*	Instrument the replacement event applies to.
perUnitReplacement	[*InstrumentQuantity*]	Instrument quantities the target instrument is replaced with.
observers	*Parties*	Observers.

Choice Archive
Controller: providers
Returns: ()
(no fields)
interface instance *I* **for** *Event*
interface instance *I* **for** *Event*

Data Types

type *T* = *Event*
Type synonym for `Event`.

1.22.3.169 Daml.Finance.Lifecycle.Rule.Claim

Templates

template *Rule*

Rule contract that allows an actor to claim effects, returning settlement instructions.

Signatory: providers

Field	Type	Description
providers	*Parties*	Providers of the claim rule. Together with the actors of the `ClaimEffect` choice the authorization requirements to upgrade the holdings being claimed have to be met.
claimers	*Parties*	Any of the parties can claim an effect.
settlers	*Parties*	Any of the parties can trigger settlement of the resulting batch.
routeProviderCid	ContractId *I*	RouteProvider used to discover settlement routes.
settlementFactoryCid	ContractId *F*	Settlement factory contract used to create a `Batch` of `Instruction`\s.
netInstructions	Bool	Configure whether netting should be enabled for quantities having the same (instrument, sender, receiver).

Choice Archive
Controller: providers
Returns: ()
(no fields)
interface instance *I* **for** *Rule*

Data Types

type *T* = *Rule*
Type synonym for `Rule`.

1.22.3.170 Daml.Finance.Lifecycle.Rule.Distribution

Templates

template *Rule*

Rule contract that defines the distribution of units of an instrument for each unit of a target instrument (e.g. share or cash dividends).

Signatory: providers

Field	Type	Description
providers	*Parties*	Providers of the distribution rule.
lifecycler	Party	Party performing the lifecycling.
observers	*Parties*	Observers of the distribution rule.
id	*Id*	Identifier for the rule contract.
description	Text	Textual description.

Choice Archive
Controller: providers
Returns: ()
(no fields)

interface instance *I* **for** *Rule*

Data Types

type *T* = *Rule*
Type synonym for `Rule`.

1.22.3.171 Daml.Finance.Lifecycle.Rule.Replacement

Templates

template *Rule*

Rule contract that defines the replacement of units of an instrument with a basket of other instruments (e.g. stock merger).

Signatory: providers

Field	Type	Description
providers	*Parties*	Providers of the replacement rule.
lifecycler	Party	Party performing the lifecycling.
observers	*Parties*	Observers.
id	*Id*	Identifier for the rule contract.
description	Text	Textual description.

Choice Archive
Controller: providers
Returns: ()
(no fields)
interface instance *I* **for** *Rule*

Data Types

type *T* = *Rule*
Type synonym for `Rule`.

1.22.3.172 Daml.Finance.Lifecycle.Rule.Util

Data Types

data *Pending*

Type used to record pending payments.

Pending

Field	Type	Description
instrument	*InstrumentKey*	
amount	Decimal	

instance Eq *Pending*

instance Show *Pending*

Functions

mergeConsumedAndProduced : [*InstrumentQuantity*] -> [*InstrumentQuantity*] -> [*Pending*]
Merge consumed and produced instruments into a list of pending settlements. This will only reproduce instrument and quantity, not tag or time.

splitPending : [*Pending*] -> ([*InstrumentQuantity*], [*InstrumentQuantity*])
Map pending settlements into corresponding instrument quantities and split them into consumed and produced. Pending items with an amount of `0.0` are discarded.

net : [*Pending*] -> [*Pending*]
Net pending payments on the same instrument (regardless of tags).

1.22.3.173 Daml.Finance.Settlement.Batch

Templates

template *Batch*

Allows you to atomically settle a set of settlement `Step`.

Signatory: requestors

Field	Type	Description
requestors	*Parties*	Parties requesting the settlement.
settlers	*Parties*	Any of the parties can trigger the settlement.
id	*Id*	Batch identifier.
description	Text	Batch description.
contextId	Optional *Id*	Identifier to link a batch to a context (e.g. the `Effect` it originated from).
routedStepsWithInstructionId	[(*RoutedStep*, *Id*)]	The settlement `RoutedStep`\s and the identifiers of the corresponding `Instruction`\s.
settlementTime	Optional Time	Settlement time (if any).

Choice Archive
Controller: requestors
Returns: ()
(no fields)
interface instance *I* **for** *Batch*

Data Types

type *T* = *Batch*
Type synonym for `Batch`.

1.22.3.174 Daml.Finance.Settlement.Factory

Templates

template *Factory*

Factory template that implements the `Factory` interface. It is used to create a set of settlement `Instruction`\s, and a `Batch` to atomically settle them.

Signatory: provider

Field	Type	Description
provider	Party	Party providing the facility.
observers	*Parties*	Observers.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *F* **for** *Factory*

1.22.3.175 Daml.Finance.Settlement.Hierarchy

Data Types

data *Hierarchy*

Data type that describes a hierarchical account structure among multiple parties for holdings on an instrument.

Hierarchy

Field	Type	Description
rootCustodian	Party	Root custodian of the instrument.
pathsToRootCustodian	[[Party]]	Paths from "leaf" owners to the root custodian of the instrument.

instance Eq *Hierarchy*

instance Show *Hierarchy*

Functions

unfoldStep : *Hierarchy* -> *Step* -> Optional [*RoutedStep*]

1.22.3.176 Daml.Finance.Settlement.Instruction

Templates

template *Instruction*

Instruction is used to settle a single settlement `Step`. In order to settle the instruction,

the sender must allocate a suitable holding
the receiver must define the receiving account

Signatory: requestors, signedSenders, signedReceivers

Field	Type	Description
requestors	*Parties*	Parties requesting the settlement.
settlers	*Parties*	Any of the parties can trigger the settlement.
batchId	*Id*	Trade identifier.
id	*Id*	Instruction identifier.
routedStep	*RoutedStep*	Routed settlement step.
settlementTime	Optional Time	Settlement time (if any).
allocation	*Allocation*	Allocation from the sender.
approval	*Approval*	Approval from the receiver.
signedSenders	*Parties*	Additional signatories, used to collect authorization.
signedReceivers	*Parties*	Additional signatories, used to collect authorization.
observers	*PartiesMap*	Observers.

Choice Archive
Controller: requestors, signedSenders, signedReceivers
Returns: ()
(no fields)
interface instance *I* **for** *Instruction*
interface instance *I* **for** *Instruction*

Data Types

type *T* = *Instruction*
Type synonym for `Instruction`.

1.22.3.177 Daml.Finance.Settlement.RouteProvider.IntermediatedStatic

Templates

template *IntermediatedStatic*

Template which implements the `RouteProvider` interface. It is used to discover the settlement route for each settlement `Step`, i.e., `RoutedSteps`\s. For each instrument to settle as part of the batch, a hierarchy of intermediaries is specified in `paths`. This hierarchy is used to generate the `RoutedStep`\s.

Signatory: provider

Field	Type	Description
provider	Party	Party providing the facility.
observers	*Parties*	Observers.
paths	Map Text *Hierarchy*	Hierarchical paths used to settle holding transfers. A path is specified for each instrument label.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *I* **for** *IntermediatedStatic*

1.22.3.178 Daml.Finance.Settlement.RouteProvider.SingleCustodian

Templates

template *SingleCustodian*

Template which implements the `RouteProvider` interface. It is used to transform each settlement `Step` into a `RoutedStep` using a single custodian.

Signatory: provider

Field	Type	Description
provider	Party	Party providing the facility.
observers	*Parties*	Observers.
custodian	Party	The custodian to be used to route each `Step`.

Choice Archive
Controller: provider
Returns: ()
(no fields)
interface instance *I* **for** *SingleCustodian*

1.22.3.179 Daml.Finance.Util.Common

Functions

notNull : [a] -> Bool
Checks if the input list is not empty.

sortAndGroupOn : Ord k => (a -> k) -> [a] -> [[a]]
Like `List.groupOn`, but sorts the list first.

1.22.3.180 Daml.Finance.Util.Date.Calendar

Functions

merge : [*HolidayCalendarData*] -> *HolidayCalendarData*
Merge multiple holiday calendars into a single one. `id`\s are concatenated by `,`.

isHoliday : *HolidayCalendarData* -> Date -> Bool
Check if Date is a holiday.

isBusinessDay : *HolidayCalendarData* -> Date -> Bool
Check if Date is a business day.

nextBusinessDay : *HolidayCalendarData* -> Date -> Date
Get next business day.

previousBusinessDay : *HolidayCalendarData* -> Date -> Date
Get previous business day.

nextOrSameBusinessDay : *HolidayCalendarData* -> Date -> Date
Get next or same business day.

previousOrSameBusinessDay : *HolidayCalendarData* -> Date -> Date
Get previous or same business day.

nextSameOrLastInMonthBusinessDay : *HolidayCalendarData* -> Date -> Date
Get next or same business day if before end of month. Otherwise get last business day in month.

previousSameOrFirstInMonthBusinessDay : *HolidayCalendarData* -> Date -> Date
Get previous or same business day if before end of month. Otherwise get first business day in month.

addBusinessDays : *HolidayCalendarData* -> Int -> Date -> Date
Add business days to a Date.

adjustDate : *HolidayCalendarData* -> *BusinessDayConventionEnum* -> Date -> Date
Adjust date according to the given business day convention.

1.22.3.181 Daml.Finance.Util.Date.DayCount

Functions

calcDcf : *DayCountConventionEnum* -> Date -> Date -> Decimal
Calculates the day count fraction given the correponding convention. Currently 30E360 is not supported as we do not want to expose the maturity date of the product as an additional parameter.

calcPeriodDcf : *DayCountConventionEnum* -> *SchedulePeriod* -> Bool -> Date -> *ScheduleFrequency* -> Decimal
Calculate day count fraction for a schedule period. It takes the following parameters:
DayCountConventionEnum: to specify which day count convention should be used
SchedulePeriod: the schedule period for which the day count fraction should be calculated
Bool: Whether day count fraction should be calculated on adjusted dates (if False: unadjusted dates)
Date: The maturity date of the instrument
Frequency: the frequency of the schedule period

calcPeriodDcfActActIsda : *SchedulePeriod* -> Bool -> Date -> Decimal
Calculate Actual Actual day count fraction according to the ISDA method.

calcPeriodDcfActActIsma : *SchedulePeriod* -> Bool -> Date -> *ScheduleFrequency* -> Decimal
Calculate Actual Actual day count fraction according to the ISMA method.

calcDcfActActAfb : Date -> Date -> Decimal

calcDcfAct360 : Date -> Date -> Decimal

calcDcfAct365Fixed : Date -> Date -> Decimal

calcDcfAct365L : Date -> Date -> Decimal

calcDcf30360 : Date -> Date -> Decimal

calcDcf30360Icma : Date -> Date -> Decimal

calcDcf30E360 : Bool -> Date -> Date -> Decimal
Calculate 30E/360 day count fraction.

1.22.3.182 Daml.Finance.Util.Date.RollConvention

Functions

next : Date -> *Period* -> *RollConventionEnum* -> Date
Get next periodic (daily `D` and weekly `W` not supported) date according to a given roll convention.

previous : Date -> *Period* -> *RollConventionEnum* -> Date
Get previous periodic (daily `D` and weekly `W` not supported) date according to a given roll convention.

addPeriod : Date -> *Period* -> Date
Add period to given date.

1.22.3.183 Daml.Finance.Util.Date.Schedule

Functions

createSchedule : [*HolidayCalendarData*] -> *PeriodicSchedule* -> *Schedule*
Generate schedule from a periodic schedule.

1.22.3.184 Daml.Finance.Util.Disclosure

This module contains default implementations for the methods of the `Disclosure` interface. These are used across multiple templates in the library.

Functions

setObserversImpl : (HasCreate t, HasField "observers" t *PartiesMap*, HasFromInterface t *I*, HasToInterface t *I*, HasInterfaceTypeRep i, HasToInterface i *I*, HasToInterface t i) => t -> Optional (*PartiesMap* -> ContractId i -> Update (ContractId *I*)) -> *SetObservers* -> Update (ContractId *I*)
Default implementation for `setObservers`. The `refUpdate` argument is used to update the corresponding contract `Reference` and can be set to `None` if your template does not have an accompanying `Reference` contract.

addObserversImpl : (HasCreate t, HasField "observers" t *PartiesMap*, HasFromInterface t *I*, HasToInterface t *I*, HasInterfaceTypeRep i, HasToInterface i *I*, HasToInterface t i) => t -> Optional (*PartiesMap* -> ContractId i -> Update (ContractId *I*)) -> *AddObservers* -> Update (ContractId *I*)
Default implementation for `addObservers`. The `refUpdate` argument is used to update the corresponding contract `Reference` and can be set to `None` if your template does not have an accompanying `Reference` contract.

removeObserversImpl : (HasCreate t, HasField "observers" t *PartiesMap*, HasFromInterface t *I*, HasToInterface t *I*, HasInterfaceTypeRep i, HasToInterface i *I*, HasToInterface t i) => t -> Optional (*PartiesMap* -> ContractId i -> Update (ContractId *I*)) -> ContractId *I* -> *RemoveObservers* -> Update (Optional (ContractId *I*))
Default implementation for `removeObservers`. The `refUpdate` argument is used to update the corresponding contract `Reference` and can be set to `None` if your template does not have an accompanying `Reference` contract.

1.23 Intro

1.23.1 Introduction to Canton

Canton is a Daml ledger interoperability protocol. Parties which are hosted on different participant nodes can transact using smart-contracts written in Daml and the Canton protocol. The Canton protocol allows to connect different Daml ledgers into a single virtual global ledger. Daml, as the smart contract language, defines who is entitled to see and who is authorized to change any given contract. The Canton synchronization protocol enforces these visibility and authorization rules, and ensures that the data is shared reliably with very high levels of privacy, even in the presence of malicious actors. The Canton network can be extended without friction with new parties, ledgers, and applications building on other applications. Extensions require neither a central managing entity nor consensus within the global network.

Canton faithfully implements the authorization and privacy requirements set out by Daml for its transactions.

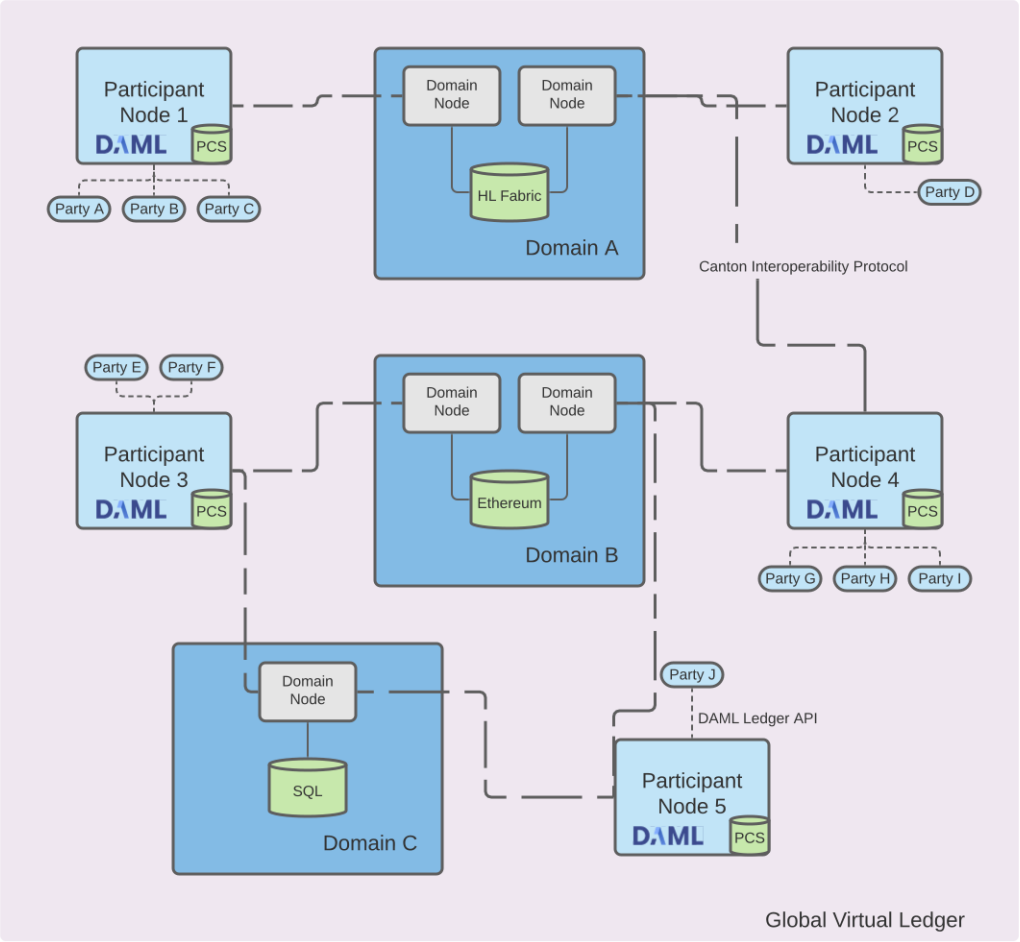


Fig. 9: Parties are hosted on participant nodes. Applications connect as parties to their participant node using the Ledger API. The participant node runs the Daml interpreter for the locally installed Daml smart contract code and stores the smart contracts in the *private contract store (PCS)*. The participants connect to domains and synchronise their state with other participants by exchanging Canton protocol messages with other participants leveraging the domain services. The use of the Canton protocol creates a virtual global ledger.

Canton is written in Scala and runs as a Java process against a database (currently H2 and Postgres). Canton is *easy to set up*, *easy to develop on* and is *easy to operate safe and securely*.

1.23.2 Overview and Assumptions

In this section, we provide an overview of the Canton architecture, illustrate the high-level flows, entities (defining trust domains) and components. We then state the trust assumptions we make on the different entities, and the assumptions on communication links.

Canton is designed to fulfill its *high-level requirements* and we assume that the reader is familiar with the Daml language and the *hierarchical transactions* of the DA ledger model.

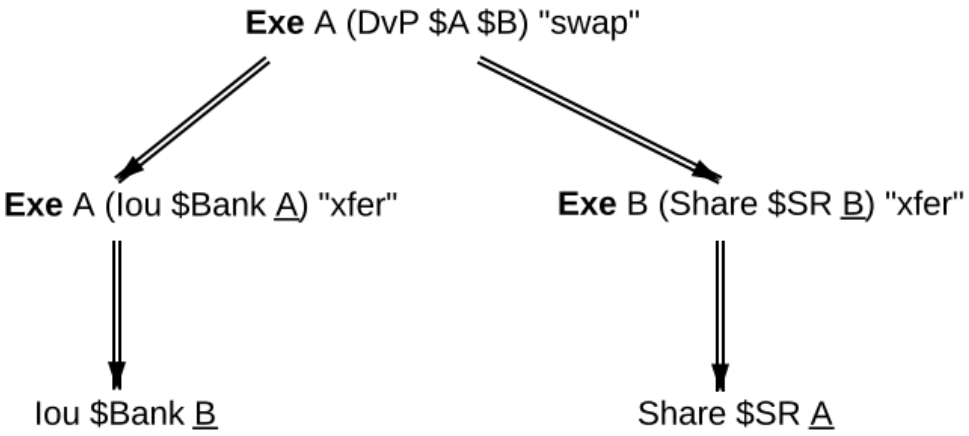
1.23.2.1 Canton 101

A Basic Example

We will use a simple delivery-versus-payment (DvP) example to provide some background on how Canton works. Alice and Bob want to exchange an IOU given to Alice by a bank for some shares that Bob owns. We have four parties: Alice (aka A), Bob (aka B), a Bank and a share registry SR. There are also three types of contracts:

1. an Iou contract, always with Bank as the backer
2. a Share contract, always with SR as the registry
3. a DvP contract between Alice and Bob

Assume that Alice has a swap choice on a DvP contract instance that exchanges an Iou she owns for a Share that Bob has. We assume that the Iou and Share contract instances have already been allocated in the DvP. Alice wishes to commit a transaction executing this swap choice; the transaction has the following structure:



Transaction Processing in Canton

In Canton, committing the example transaction consists of two steps:

1. Alice's participant prepares a **confirmation request** for the transaction. The request provides different views on the transaction; participants see only the subtransactions exercising, fetching or creating contracts on which their parties are stakeholders (more precisely, the subtransactions where these parties are *informees*). The views for the DvP, and their recipients, are shown in the figure below. Alice's participant submits the request to a **sequencer**, who orders all confirmation requests on a Canton domain; whenever two participants see the same two requests, they will see them according to this sequencer order. The sequencer has only two functions: ordering messages and delivering them to their stated recipients. The message contents are encrypted and not visible to the sequencer.
2. The recipients then check the validity of the views that they receive. The validity checks cover three aspects:
    1. validity as *defined* in the DA ledger model: *consistency*, (mainly: no double spends), *conformance* (the view is a result of a valid Daml interpretation) and *authorization* (guaranteeing

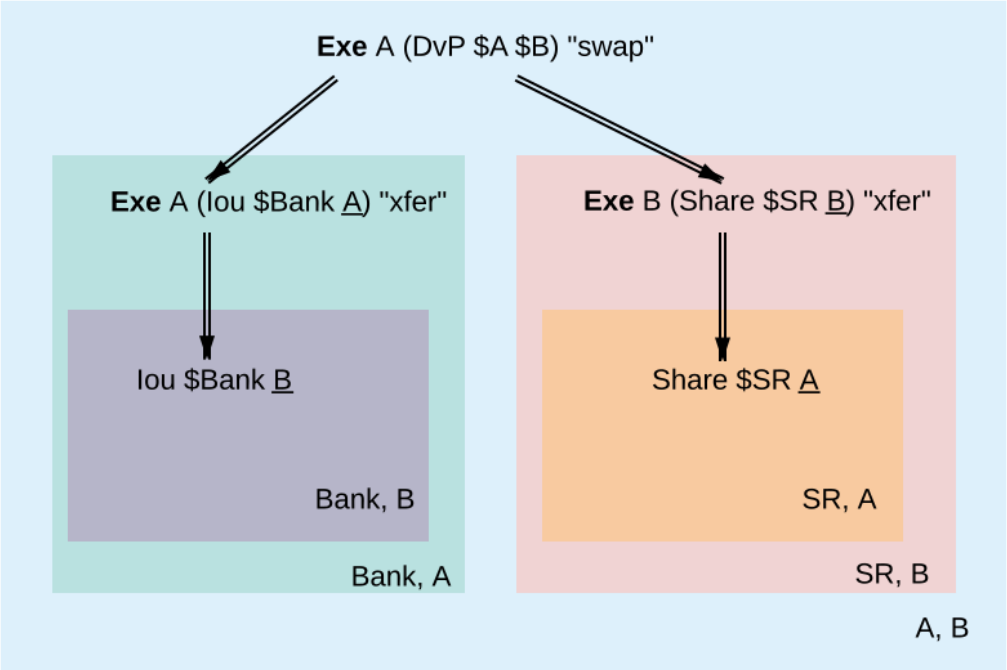


Fig. 10: Views in the transaction; each box represents a transaction part visible to the participants in its bottom-right corner. A participant might receive several views, some of which can be nested.

that the actors and submitters are allowed to perform the view's action)
2. authenticity (guaranteeing that the actors and submitters are who they claim to be).
3. transparency (guaranteeing that participants who should be notified get notified).

Conformance, authorization, authenticity and transparency problems only arise due to submitter malice. Consistency problems can arise with no malice. For example, the Iou that is to be transferred to Bob might simply have already been spent (assuming that we do not use the locking technique in Daml). Based on the check's result, a subset of recipients, called **confirmers** then prepares a (positive or negative) **confirmation response** for each view separately. A **confirmation policy** associated with the request specifies which participants are confirmers, given the transaction's informees.

The confirmers send their responses to a **mediator**, another special entity that aggregates the responses into a single decision for the entire confirmation request. The mediator serves to hide the participants' identities from each other (so that Bank and SR do not need to know that they are part of the same transaction). Like the sequencer, the mediator does not learn the transactions' contents. Instead, Alice's participant, in addition to sending the request, also simultaneously notifies the mediator about the informees of each view. The mediator receives a version of the transaction where only the informees of a view are visible and the contents blinded, as conceptually visualized in the diagram below.

From this, the mediator derives which (positive) confirmation responses are necessary in order to decide the confirmation request as **approved**.

Requests submitted by malicious participants can contain bogus views. As participants can see only parts of requests (due to privacy reasons), upon receiving an approval for a request, each participant locally filters out the bogus views that are visible to it, and **accepts** all remaining valid views of an approved confirmation request. Under the confirmation policy's trust assumptions, the protocol ensures that the local decisions of honest participants match for all views that they jointly see. The protocol thus provides a virtual shared ledger between the participants, whose transactions consist of such valid views. Once approved, the accepted views

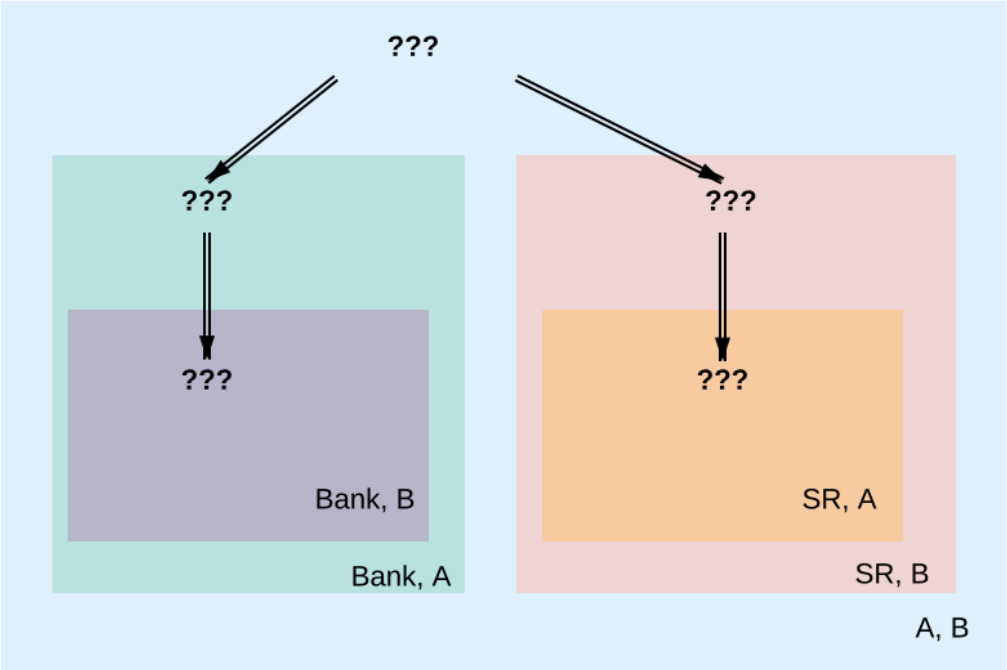


Fig. 11: In the informee tree for the mediator, all transaction contents are blinded.

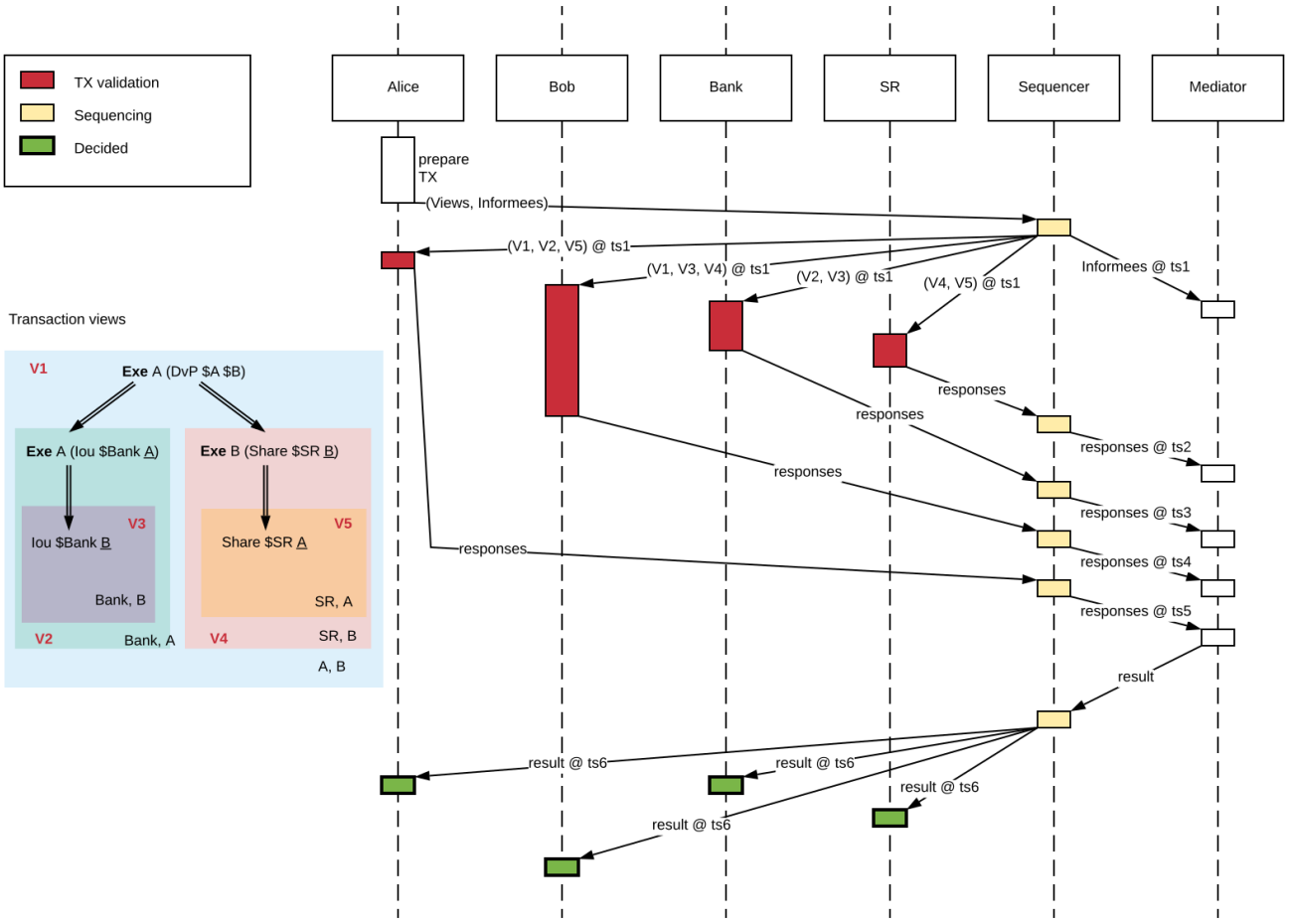
are **final**, i.e., they will never be removed from the participants' records or the virtual ledger.

We can represent the confirmation workflow described above by the following message sequence diagram, assuming that each party in the example runs their own participant node.

The sequencer and the mediator, together with a so-called **topology manager** (described shortly), constitute a **Canton domain**. All messages within the domain are exchanged over the sequencer, which ensures a **total order** between all messages exchanged within a domain.

The total ordering ensures that participants see all confirmation requests and responses in the same order. The Canton protocol additionally ensures that all non-Byzantine (i.e. not malicious or compromised) participants see their shared views (such as the exercise of the Iou transfer, shared between the participants of Bank and A) in the same order, even with Byzantine submitters. This has the following implications:

1. The correct confirmation response for each view is always uniquely determined, because Daml is deterministic. However, for performance reasons, we allow occasional incorrect negative responses, when participants start behaving in a Byzantine fashion or under contention. The system provides the honest participants with evidence of either the correctness of their responses or the reason for the incorrect rejections.
2. The global ordering creates a (virtual) **global time** within a domain, measured at the sequencer; participants learn that time has progressed whenever they receive a message from the sequencer. This global time is used for detecting and resolving conflicts and determining when timeouts occur. Conceptually, we can therefore speak of a step happening at several participants simultaneously with respect to this global time, although each participant performs this step at a different physical time. For example, in the above *message sequence diagram*, Alice, Bob, the Bank, and the share registry's participants receive the confirmation request at different physical times, but conceptually this happens at the timestamp *ts1* of the global time, and similarly for the result message at timestamp *ts6*.

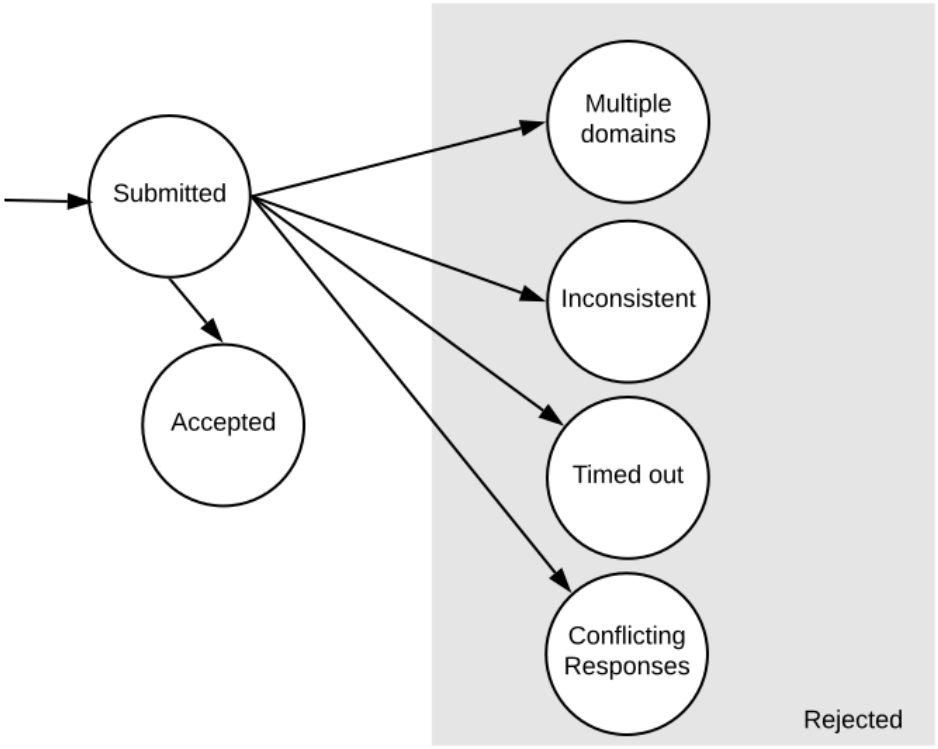

TX validation
Sequencing
Decided
Alice
Bob
Bank
SR
Sequencer
Mediator
prepare TX
(Views, Informees)
(V1, V2, V5) @ ts1
(V1, V3, V4) @ ts1
(V2, V3) @ ts1
(V4, V5) @ ts1
Informees @ ts1
responses
responses
responses @ ts2
responses
responses @ ts3
responses
responses @ ts4
responses @ ts5
result
result @ ts6
result @ ts6
result @ ts6
result @ ts6
Transaction views
V1
Exe A (DvP $A $B)
Exe A (Iou $Bank A)
Exe B (Share $SR B)
V3
Iou $Bank B
Bank, B
V5
Share $SR A
SR, A
V2
Bank, A
V4
SR, B
A, B

In this document, we focus on the basic version of Canton, with just a single domain. Canton also supports connecting a participant to multiple domains and transferring contracts between domains (see *composability*).

As mentioned in the introduction, the main challenges for Canton are reconciling integrity and privacy concerns while ensuring progress with the confirmation-based design, given that parties might be overloaded, offline, or simply refusing to respond. The main ways we cope with this problem are as follows:

> We use timeouts: if a transaction's validity cannot be determined after a timeout (which is a domain-wide constant), the transaction is rejected.
> If a confirmation request times out, the system informs the participant submitting the request on which participants have failed to send a confirmation response. This allows the submitting participant to take out of band actions against misbehaviour.
> Flexible confirmation policies: To offer a trade-off between trust, integrity, and liveness, we allow Canton domains to choose their *confirmation policies*. Confirmation policies specify which participants need to confirm which views. This enables the mediator to determine the sufficient conditions to declare a request approved. Of particular interest is the *VIP confirmation policy*, applicable to transactions which involve a trusted (VIP) party as an informee on every action. An example of a VIP party is a market operator. The policy ensures ledger validity assuming the VIP party's participants behave correctly; incorrect behavior can still be detected and proven in this case, but the fallout must be handled outside of the system. Another important policy is the signatory confirmation policy, in which all signatories and actors are required to confirm. This requires a lower level of trust compared to the VIP confirmation policy sacrificing liveness when participants hosting signatories or actors are unresponsive. Another policy (being deprecated) is the *full confirmation policy*, in which all informees are required to confirm. This requires the lowest level of trust, but sacrifices liveness when some of the involved participants are unresponsive.
> In the future, we will support attestators, which can be thought of as on-demand VIP participants. Instead of constructing Daml models so that VIP parties are informees on every action, attestators are only used on-demand. The participants who wish to have the transaction committed must disclose sufficient amount of history to provide the attestator with unequivocal evidence of a subtransaction's validity. The attestator's statement then substitutes the confirmations of the unresponsive participants.

The following image shows the state transition diagram of a confirmation request; all states except for Submitted are final.



A confirmation request can be rejected for several reasons:

Multiple domains The transaction tried to use contracts created on different Canton domains. Multi-domain transactions are currently not supported.

Timeout Insufficient confirmations have been received within the timeout window to declare the transaction as accepted according to the confirmation policy. This happens due to one of the involved participants being unresponsive. The request then times out and is aborted. In the future, we will add a feature where aborts can be triggered by the submitting party, or anyone else who controls a contract in the submitted transaction. The aborts still have to happen after the timeout, but are not mandatory. Additionally, attestators can be used to supplant the confirmations from the unresponsive participants.

Inconsistency It conflicts with an earlier pending request, i.e., a request that has neither been approved nor rejected yet. Canton currently implements a simple **pessimistic** *conflict resolution policy*, which always fails the later request, even if the earlier request itself gets rejected at some later point.

Conflicting responses Conflicting responses were received. In Canton, this only happens when one of the participants is Byzantine.

Conflict Detection

Participants detect conflicts between concurrent transactions by locking the contracts that a transaction consumes. The participant locks a contract when it receives the confirmation request of a transaction that archives the contract. The lock indicates that the contract is possibly archived. When the mediator's decision arrives later, the contract is unlocked again - and archived if the transaction was approved. When a transaction wants to use a possibly archived contract, then this transaction will be rejected in the current version of Canton. This design decision is based on the optimistic assumption that transactions are typically accepted; the later conflicting transaction can therefore be pessimistically rejected.

The next three diagrams illustrate locking and pessimistic rejections using the *counteroffer* example from the DA ledger model. There are two transactions and three parties and every party runs their own participant node.

> The painter *P* accepts *A*'s *Counteroffer* in transaction *tx1*. This transaction consumes two contracts:
> - The Iou between *A* and the *Bank*, referred to as *c1*.
> - The *Counteroffer* with stakeholders *A* and *P*, referred to as *c2*.
>
> The created contracts (the new Iou and the *PaintAgree*ment) are irrelevant for this example.
> Suppose that the *Counteroffer* contains an additional consuming choice controlled by *A*, e.g., Alice can retract her *Counteroffer*. In transaction *tx2*, *A* exercises this choice to consume the *Counteroffer c2*.

Since the messages from the sequencer synchronize all participants on the (virtual) global time, we may think of all participants performing the locking, unlocking, and archiving simultaneously.

In the first diagram, the sequencer sequences *tx1* before *tx2*. Consequently, *A* and the *Bank* lock *c1* when they receive the confirmation request, and so do *A* and *P* for *c2*. So when *tx2* later arrives at *A* and *P*, the contract *c2* is locked. Thus, *A* and *P* respond with a rejection and the mediator follows suit. In contrast, all stakeholders approve *tx1*; when the mediator's approval arrives at the participants, each participant archives the appropriate contracts: *A* archives *c1* and *c2*, the *Bank* archives *c1*, and *P* archives *c2*.

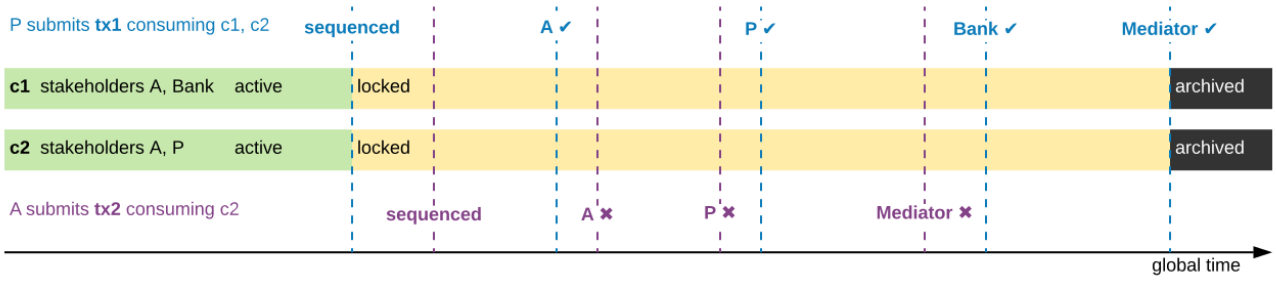


Fig. 12: When two transactions conflict while they are in flight, the later transaction is always rejected.

The second diagram shows the scenario where *A*'s retraction is sequenced before *P*'s acceptance of the *Counteroffer*. So *A* and *P* lock *c2* when they receive the confirmation request for *tx2* from the sequencer and later approve it. For *tx1*, *A* and *P* notice that *c2* is possibly archived and therefore reject *tx1*, whereas everything looks fine for the *Bank*. Consequently, the *Bank* and, for consistency, *A* lock *c1* until the mediator sends the rejection for *tx1*.

Note: In reality, participants approve each view individually rather than the transaction as a whole. So *A* sends two responses for *tx1*: An approval for *c1*'s archival and a rejection for *c2*'s archival. The diagrams omit this technicality.

The third diagram shows how locking and pessimistic rejections can lead to incorrect negative responses. Now, the painter's acceptance of *tx1* is sequenced before Alice's retraction like in the *first diagram*, but the Iou between *A* and the *Bank* has already been archived earlier. The painter receives only the view for *c2*, since *P* is not a stakeholder of the Iou *c1*. Since everything looks fine, *P* locks *c2* when the confirmation request for *tx1* arrives. For consistency, *A* does the same, although *A* already knows that the transaction will fail because *c1* is archived. Hence, both *P* and *A* reject *tx2* because it

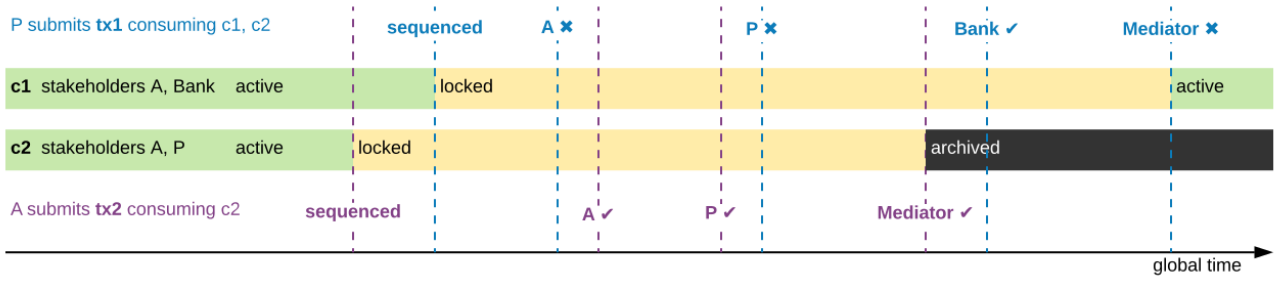


Fig. 13: Transaction *tx2* is now submitted before *tx1*. The consumed contract *c1* remains locked by the rejected transaction until the mediator sends the result message.

tries to consume the locked contract *c2*. Later, when *tx1*'s rejection arrives, *c2* becomes active again, but the transaction *tx2* remains rejected.

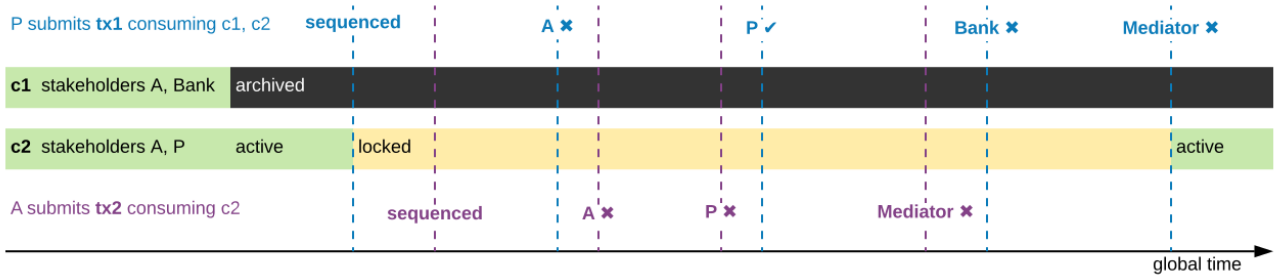


Fig. 14: Even if the earlier transaction *tx1* is rejected later, the later conflicting transaction *tx2* remains rejected and the contract remains locked until the result message.

Time in Canton

The connection between time in Daml transactions and the time defined in Canton is explained in the respective ledger model section on time.

The respective section introduces *ledger time* and *record time*. The *ledger time* is the time the participant (or the application) chooses when computing the transaction prior to submission. We need the participant to choose this time as the transaction is pre-computed by the submitting participant and this transaction depends on the chosen time. The *record time* is assigned by the sequencer when registering the confirmation request (initial submission of the transaction).

There is only a bounded relationship between these times, ensuring that the *ledger time* must be in a pre-defined bound around the *record time*. The tolerance (`max_skew`) is defined on the domain as a domain parameter, known to all participants

```
canton.domains.mydomain.parameters.ledger-time-record-time-tolerance
```

The bounds are symmetric in Canton, so `min_skew` equals `max_skew`, equal to above parameter.

Note: Canton does not support querying the time model parameters via the ledger API, as the time model is a per domain property and this cannot be properly exposed on the respective ledger API

endpoint.

Checking that the *record time* is within the required bounds is done by the validating participants and is visible to everyone. The sequencer does not know what was timestamped and therefore doesn't perform this validation.

Therefore, a submitting participant cannot control the output of a transaction depending on *record time*, as the submitting participant does not know exactly the point in time when the transaction will be timestamped by the sequencer. But the participant can guarantee that a transaction will either be registered before a certain record time, or the transaction will fail.

Subtransaction privacy

Canton splits a Daml transaction into views, as described above under *transaction processing*. The submitting participant sends these views via the domain's sequencer to all involved participants on a need-to-know basis. This section explains how the views are encrypted, distributed, and stored so that only the intended recipients learn the contents of the transaction.

In the *above DvP example*, Canton creates a view for each node, as indicated by the boxes with the different colors. Canton captures this hierarchical view structure in a Merkle-like tree. For example, the view for exercising the xfer choice conceptually looks as follows, where the hashes `0x...` commit to the contents of the hidden nodes and subtrees without revealing the content. In particular, the second leg's structure, contents, and recipients are completely hidden in the hash `0x1210...`.

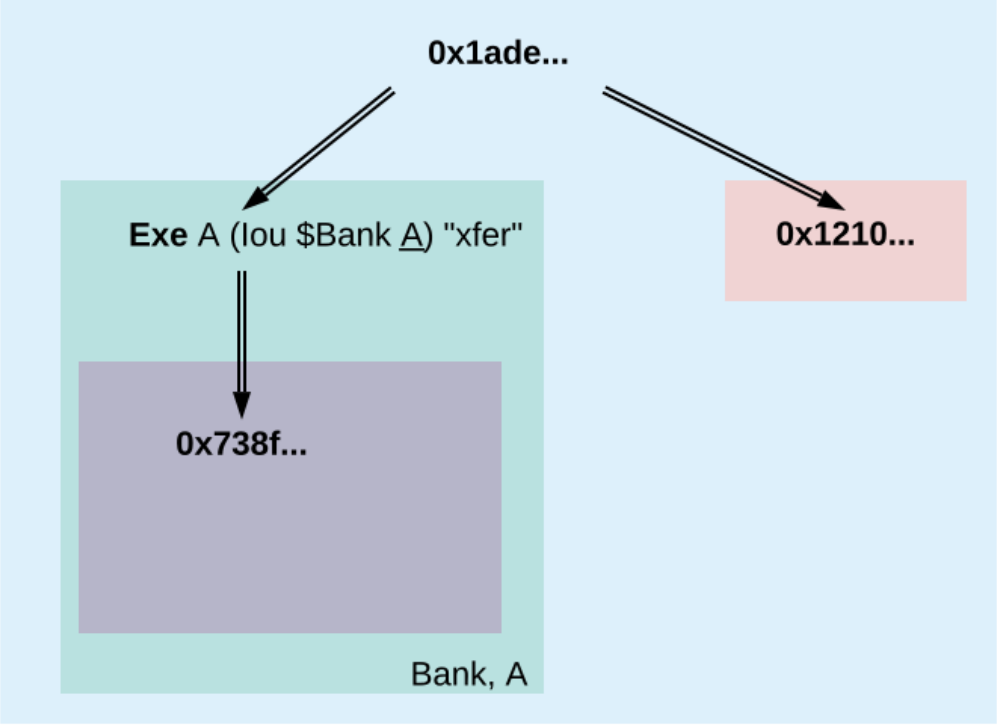


Fig. 15: Idealized Merkle tree for the view that exercises the xfer choice on Alice's Iou.

The subview that creates the transferred Iou has a similar structure, except that the hash `0x738f...` is now unblinded into the view content and the parent view's **Exercise** action is represented by its hash `0x8912...`

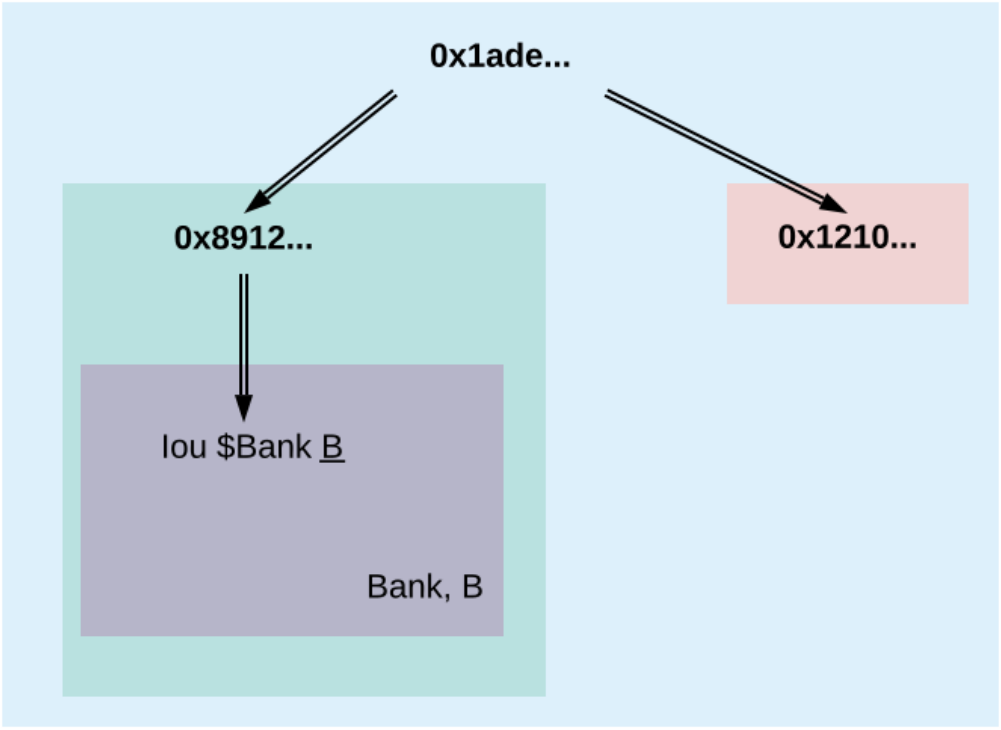


Fig. 16: Idealized Merkle tree for the view that creates Bob's new Iou.

Using the hashes, every recipient can correctly reconstruct their projection of the transaction from the views they receive.

As illustrated in the *confirmation workflow*, the submitting participant sends the views to the participants hosting an informee or witness of a view's actions. This ensures **subtransaction privacy** as a participant receives only the data for the witnesses it hosts, not all of the transaction. Each Canton participant persists all messages it receives from the sequencer, including the views.

Moreover, Canton hides the transaction contents from the domain too. To that end, the submitting participant encrypts the views using the following hybrid encryption scheme:

1. It generates cryptographic randomness for the transaction, the transaction seed. From the transaction seed, a view seed is derived for each view following the hierarchical view structure, using a pseudo-random function. In the DvP example, a view seed $seed_0$ for the action at the top is derived from the transaction seed. The seed $seed_1$ for the view that exercises the xfer choice is derived from the parent view's seed $seed_0$, and similarly the seed $seed_2$ for the view that creates Bob's IOU is derived from $seed_1$.
2. For each view, it derives a symmetric encryption key from the view seed using a key derivation function. For example, the symmetric key for the view that creates Bob's IOU is derived from $seed_2$. Since the transaction seed is fresh for every submission and all derivations are cryptographically secure, each such symmetric key is used only once.
3. It encrypts the serialization of each view's Merkle tree with the symmetric key derived for this view. The view seed itself is encrypted with the public key of each participant hosting an informee of the view. The encrypted Merkle tree and the encryptions of the view seed form the data that is sent via the sequencer to the recipients.

Note: The view seed is encrypted only with the public key of the participants that host an informee, while the encrypted Merkle tree itself is also sent to participants hosting only wit-

nesses. The latter participants can nevertheless decrypt the Merkle tree because they receive the view seed of a parent view and can derive the symmetric key of the witnessed view using the derivation functions.

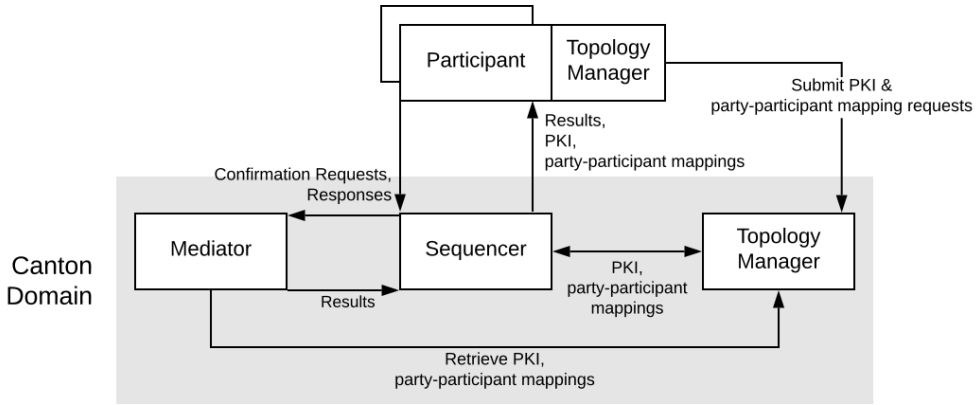
Even though the sequencer persists the encrypted views for a limited period, the domain cannot access the symmetric keys unless it knows the secret key of one of the informee participants. Therefore, the transaction contents remain confidential with respect to the domain.

1.23.2.2 Domain Entities

A Canton domain consists of three entities:

the sequencer
the mediator
and the **topology manager**, providing a PKI infrastructure, and party to participant mappings.

We call these the **domain entities**. The high-level communication channels between the domain entities are depicted below.



In general, every domain entity can run in a separate trust domain (i.e., can be operated by an independent organization). In practice, we assume that all domain entities are run by a single organization, and that the domain entities belong to a single trust domain.

Furthermore, each participant node runs in its own trust domain. Additionally, the participant may outsource a part of its identity management infrastructure, for example to a certificate authority. We assume that the participant trusts this infrastructure, that is, that the participant and its identity management belong to the same trust domain. Some participant nodes can be designated as **VIP nodes**, meaning that they are operated by trusted parties. Such nodes are important for the VIP confirmation policy.

The generic term **member** will refer to either a domain entity or a participant node.

Sequencer

We now list the high-level requirements on the sequencer.

Ordering: The sequencer provides a global total-order multicast where messages are uniquely time-stamped and the global ordering is derived from the timestamps. Instead of delivering a single message, the sequencer provides message batching, that is, a list of individual messages are submitted. All these messages get the timestamp of the batch they are contained in. Each message may have a different set of recipients; the messages in each recipient's batch are in the same order as in the sent batch.

Evidence: The sequencer provides the recipients with a cryptographic proof of authenticity for every message batch it delivers, including evidence on the order of batches.

Sender and Recipient Privacy: The recipients do not learn the identity of the submitting participant. A recipient only learns the identities of recipients on a particular message from a batch if it is itself a recipient of that message.

Mediator

The mediator's purpose is to compute the final result for a confirmation request and distribute it to the participants, ensuring that transactions are atomically committed across participants, while preserving the participants' privacy, by not revealing their identities to each other. At a high level, the mediator:

collects confirmation responses from participants,
validates them according to the Canton protocol,
computes the conclusions (approve / reject / timed out) according to the confirmation policy, and
sends the result message.

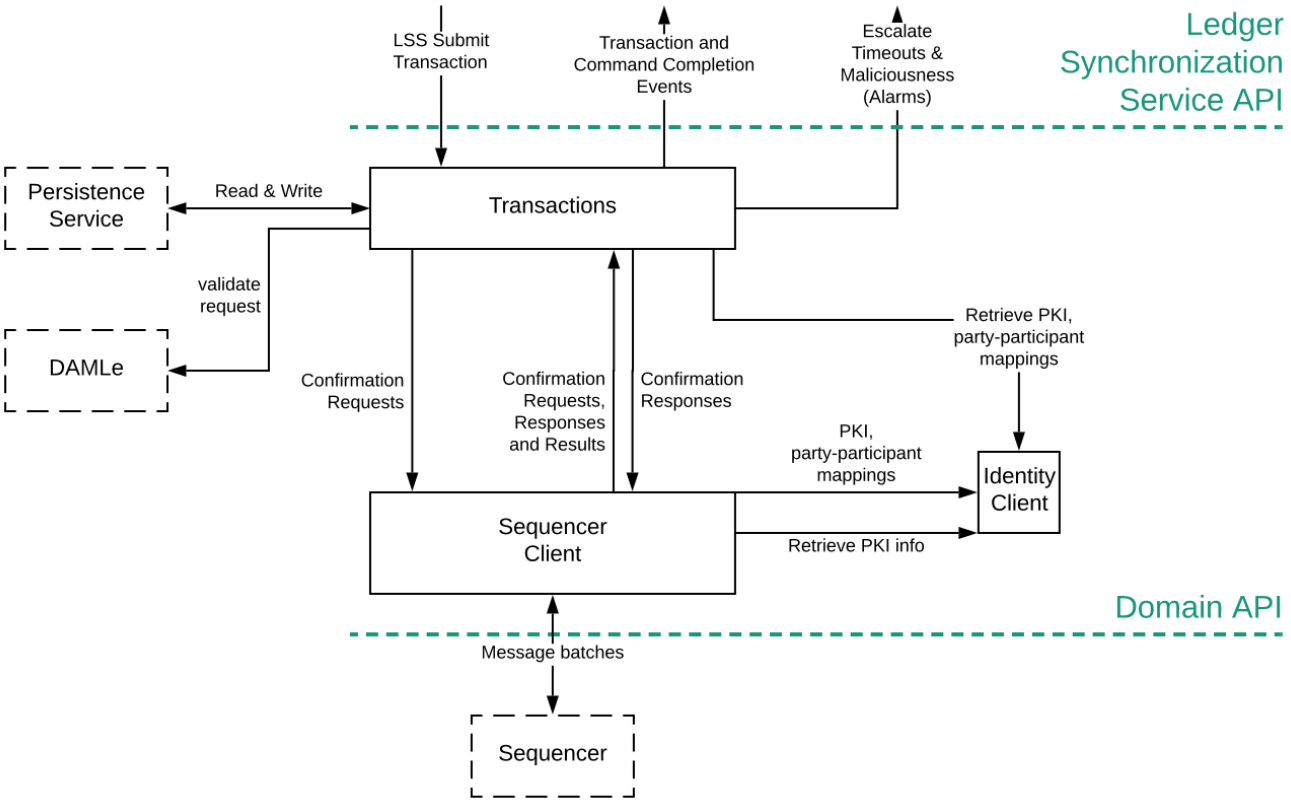
Additionally, for auditability, the mediator persists every received message (containing informee information or confirmation responses) in long term storage and allows an auditor to retrieve messages from this storage.

Topology Manager

The topology manager allows participants to join and leave the Canton domain, and to register, revoke and rotate public keys. It knows the parties **hosted** by a given participant. It defines the **trust level** of each participant. The trust level is either **ordinary** or **VIP**. A VIP trust level indicates that the participant is trusted to act honestly. A canonical example is a participant run by a trusted market operator.

1.23.2.3 Participant-internal Canton Components

Canton uses the Daml-on-X architecture, to promote code reuse. In this architecture, the participant node is broken down into a set of services, all but one of which are reused among ledger implementations. This ledger-specific service is called the Ledger Synchronization Service (LSS), which Canton implements using its protocol. This implementation is further broken down into multiple components. We now describe the interface and properties of each component. The following figure shows the interaction between the different components and the relation to the existing Ledger API's command and event services.



We next explain each component in turn.

Transactions

This is the central component of LSS within Canton. We describe the main tasks below.

Submission and Segregation: A Daml transaction has a tree-like structure. The *ledger privacy model* defines which parts of a transaction are visible to which party, and thus participant. Each recipient obtains only the subtransaction (projection) it is entitled to see; other parts of the transaction are never shared with the participant, not even in an encrypted form. Furthermore, depending on the confirmation policy, some informees are marked as confirmers. In addition to distributing the transaction projections among participants, the submitter informs the mediator about the informees and confirmers of the transaction.

Validity and Confirmations Responses: Each informee of a requested transaction performs local checks on the validity of its visible subtransaction. The informees check that their provided projection conforms to the Daml semantics, and the ledger authorization model. Additionally, they check whether the request conflicts with an earlier request that is accepted or is not yet decided. Based on this, they send their responses (one for each of their views), together with the informee information for their projection, to the mediator. When the other participants or domain entities do not behave according to the protocol (for example, not sending timely confirmation responses, or sending malformed requests), the transaction processing component raises alarms.

Confirmation Result Processing. Based on the result message from the mediator, the transaction component commits or aborts the requested transaction.

Sequencer Client

The sequencer client handles the connection to the sequencer, ensures in-order delivery and stores the cryptographic proofs of authenticity for the messages from the sequencer.

Identity Client

The identity client handles the messages coming from the domain topology manager, and verifies the validity of the received identity information changes (for example, the validity of public key delegations).

1.23.2.4 System Model And Trust Assumptions

The different sets of rules that Canton domains specify affect the security and liveness properties in different ways. In this section, we summarize the system model that we assume, as well as the trust assumptions. Some trust assumptions are dependent on the domain rules, which we indicate in the text. As specified in the *high-level requirements*, the system provides guarantees only to honestly represented parties. Hence, every party must fully trust its participant (but no other participants) to execute the protocol correctly. In particular, signatures by participant nodes may be deemed as evidence of the party's action in the transaction protocol.

System Model

We assume that pairwise communication is possible between any two system members. The links connecting the participant nodes to the sequencers and the referees are assumed to be *mostly timely*: there exists a known bound on the delay such that the overwhelming majority of messages exchanged between the participant and the sequencer are delivered within . Domain entities are assumed to have clocks that are closely synchronized (up to some known bound) for an overwhelming majority of time. Finally, we assume that the participants know a probability distribution over the message latencies within the system.

General Trust Assumptions

These assumptions are relevant for all system properties, except for privacy.

> The sequencer is trusted to correctly provide a global total-order multicast service, with evidence and ensuring the sender and recipient privacy.
> The mediator is trusted to produce and distribute all results correctly.
> The topology managers of honest participants (including the underlying public key infrastructure, if any) are operating correctly.

When a transaction is submitted with the VIP confirmation policy (in which case every action in the transaction must have at least one VIP informee), there exist an additional integrity assumption:

> All VIP stakeholders must be hosted by honest participants, i.e., participants that run the transaction protocol correctly.

We note that the assumptions can be weakened by replicating the trusted entities among multiple organization with a Byzantine fault tolerant replication protocol, if the assumptions are deemed too strong. Furthermore, we believe that with some extensions to the protocol we can make the violations of one of the above assumptions detectable by at least one participant in most cases, and often also provable to other participants or external entities. This would require direct communication between the participants, which we leave as future work.

Assumptions Relevant for Privacy

The following common assumptions are relevant for privacy:

> The private keys of honest participants are not compromised, and all certificate authorities that the honest participants use are trusted.
> The sequencer is privy to:
> 1. the submitters and recipients of all messages
> 2. the view structure of a transaction in a confirmation request, including informees and confirming parties
> 3. the confirmation responses (approve / reject / ill-formed) of confirmers.
> 4. encrypted transaction views
> 5. timestamps of all messages
>
> The sequencer is trusted with not storing messages for longer than necessary for operational procedures (e.g., delivering messages to offline parties or for crash recovery).
> The mediator is privy to:
> 1. the view structure of a transaction including informees and confirming parties, and the submitting party

2. the confirmation responses (approve / reject / ill-formed) of confirmers
3. timestamps of messages

The informees of a part of a transaction are trusted with not violating the privacy of the other stakeholders in that same part. In particular, the submitter is trusted with choosing strong randomness for transaction and contract IDs. Note that this assumption is not relevant for integrity, as Canton ensures the uniqueness of these IDs.

When a transaction is submitted with the VIP confirmation policy, every action in the transaction must have at least one VIP informee. Thus, the VIP informee is automatically privy to the entire contents of the transaction, according to the *ledger privacy model*.

Assumptions Relevant for Liveness

In addition to the general trust assumptions, the following additional assumptions are relevant for liveness and bounded liveness functional requirements on the system: bounded decision time, and no unnecessary rejections:

All the domain entities in Canton (the sequencer, the mediator, and the topology manager) are highly available.
The sequencer is trusted to deliver the messages timely and fairly (as measured by the probability distribution over the latencies).
The domain topology manager forwards all identity updates correctly.
Participants hosting confirming parties according to the confirmation policy are assumed to be highly available and responding correctly. For example in the VIP confirmation policy, only the VIP participant needs to be available whereas in the signatory policy, liveness depends on the availability of all participants that host signatories and actors.

1.23.3 Canton Demo

The Canton demo is used to demonstrate the unique Canton capabilities:

Application Composability - Add new workflows at any time to a running system
Network Interoperability - Create workflows spanning across domains
Privacy - Canton uses data minimization and only shares data on a need to know basis.
Regulatory compliance - Canton can be used to even integrate personal sensitive information directly in workflows without fear of failing to be GDPR compliant.

The demo is a thin application running on top of a setup with 5 participant nodes and 2 domains. You can run it by downloading the release package from github. Then, unpack and start it, using the following commands (or the zip equivalent)

```
tar zxvf canton-open-source-x.y.z.tar.gz
cd canton-open-source-x.y.z
bash start-demo.command
```

You need to replace `x.y.z` with the appropriate version number of the release you've downloaded. On Windows, you can just double-click the `start-demo-win.cmd` script in Windows explorer.

Note: The demo requires JavaFX. Please use a Java runtime of version 11 or greater.

If you don't want to run it yourself, you can also watch our recording.

The entire code base of the demo is included in the release package as `demo`.

1.23.4 Getting Started

Interested in Canton? This is the right place to start! You don't need any prerequisite knowledge, and you will learn:

how to install Canton and get it up and running in a simple test configuration
the main concepts of Canton
the main configuration options
some simple diagnostic commands on Canton
the basics of Canton identity management
how to upload and execute new smart contract code

1.23.4.1 Installation

Canton is a JVM application. To run it natively you need Java 11 or higher installed on your system. Alternatively Canton is available as a docker image (see *Canton docker instructions*).

Canton is platform-agnostic. For development purposes, it runs on macOS, Linux, and Windows. Linux is the supported platform for production.

Note: Windows garbles the Canton console output unless you are running Windows 10 and you enable terminal colors (e.g., by running `cmd.exe` and then executing `reg add HKCU\Console /v VirtualTerminalLevel /t REG_DWORD /d 1`).

To start, download the open source community edition latest release and extract the archive, or use the enterprise edition if you have access to it.

The extracted archive has the following structure:

```
.
├── bin
├── daml
├── dars
├── demo
├── drivers (enterprise)
├── examples
├── lib
└── ...
```

`bin`: contains the scripts for running Canton (`canton` under Unix-like systems and `canton.bat` under Windows)
`daml`: contains the source code for some sample smart contracts
`dars`: contains the compiled and packaged code of the above contracts
`demo`: contains everything needed to run the interactive Canton demo
`examples`: contains sample configuration and script files for the Canton console
`lib`: contains the Java executables (JARs) needed to run Canton

This tutorial assumes you are running a Unix-like shell.

1.23.4.2 Starting Canton

While Canton supports a daemon mode for production purposes, in this tutorial we will use its console, a built-in interactive read-evaluate-print loop (REPL). The REPL gives you an out-of-the-box interface to all Canton features. In addition, as it's built using Ammonite, you also have the full power of Scala if you need to extend it with new scripts. As such, any valid Scala expression can be typed inside the console:

```
@ Seq(1,2,3).map(_ * 2)
res1: Seq[Int] = List(2, 4, 6)
```

Navigate your shell to the directory where you extracted Canton. Then, run

```
bin/canton --help
```

to see the command line options that Canton supports. Alternatively to `bin/canton`, you can also start Canton directly with `java -jar lib/canton-*.jar`, assuming all other jar dependencies are in the `lib` folder, too.

Next, run

```
bin/canton -c examples/01-simple-topology/simple-topology.conf
```

This starts the console using the configuration file `examples/01-simple-topology/simple-topology.conf`. You will see the banner on your screen

```
  _____            _
 / ____|          | |
| |      __ _ _ __ | |_ ___  _ __
| |     / _` | '_ \| __/ _ \| '_ \
| |____| (_| | | | | || (_) | | | |
 \_____\__,_|_| |_|\__\___/|_| |_|

 Welcome to Canton!
 Type `help` to get started. `exit` to leave.
```

Type `help` to see the available commands in the console:

```
@ help
Top-level Commands
------------------
exit - Leave the console
help - Help with console commands; type help("<command>") for detailed help for
 ↪<command>

Generic Node References
-----------------------
domainManagers - All domain manager nodes (.all, .local, .remote)
..
```

You can also get help for specific Canton objects and commands:

```
@ help("participant1")
participant1
Manage participant 'participant1'; type 'participant1 help' or 'participant1 help(
 ↪"<methodName>")' for more help
```

(continues on next page)

(continued from previous page)

```
@ participant1.help("start")
start
Start the instance
```

1.23.4.3 The Example Topology

To understand the basic elements of Canton, let's briefly look at this starting configuration. It is written in the HOCON format as shown below. It specifies that you wish to run two *participant nodes*, whose local aliases are `participant1` and `participant2`, and a single *synchronization domain*, with the local alias `mydomain`. It also specifies the storage backend that each node should use (in this tutorial we're using in-memory storage), and the network ports for various services, which we will describe shortly.

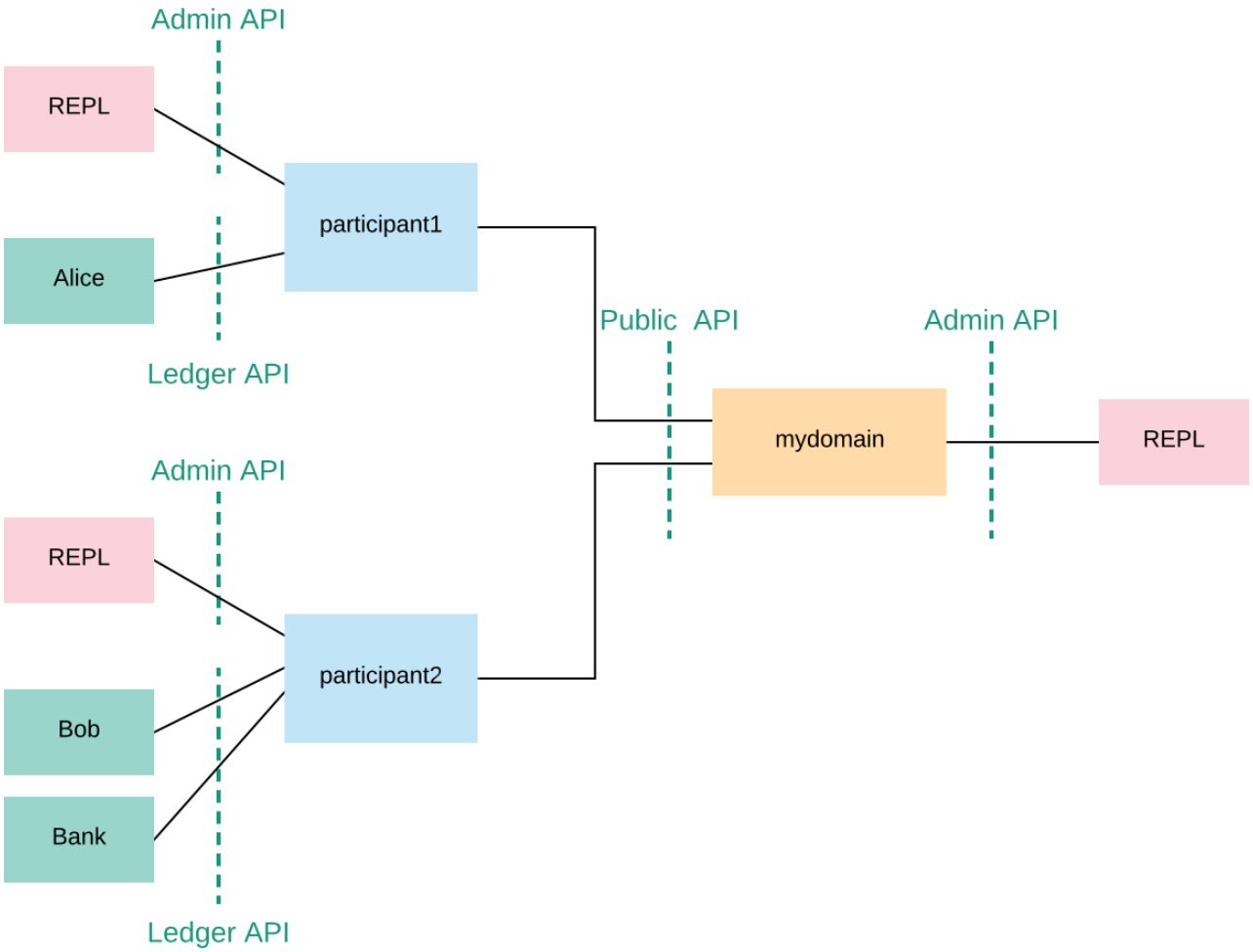
```
canton {
  participants {
    participant1 {
      storage.type = memory
      admin-api.port = 5012
      ledger-api.port = 5011
    }
    participant2 {
      storage.type = memory
      admin-api.port = 5022
      ledger-api.port = 5021
    }
  }
  domains {
    mydomain {
      storage.type = memory
      public-api.port = 5018
      admin-api.port = 5019
    }
  }
  // enable ledger_api commands for our getting started guide
  features.enable-testing-commands = yes
}
```

To run the protocol, the participants must connect to one or more synchronization domains (domains for short). To execute a *transaction* (a change that updates the shared contracts of several parties), all the parties' participant nodes must be connected to the same domain. In the remainder of this tutorial, you will construct a network topology that will enable the three parties Alice, Bob, and Bank to transact with each other, as shown here:

The participant nodes provide their parties with a Ledger API as a means to access the ledger. The parties can interact with the Ledger API manually using the console, but in practice these parties use applications to handle the interactions and display the data in a user-friendly interface.

In addition to the Ledger API, each participant node also exposes an *Admin API*. The Admin API allows the administrator (that is, you) to:

manage the participant node's connections to domains
add or remove parties to be hosted at the participant node


Admin API
REPL
Alice
participant1
Ledger API
Public API
Admin API
mydomain
REPL
Admin API
REPL
Bob
Bank
participant2
Ledger API

upload new Daml archives
configure the operational data of the participant, such as cryptographic keys
run diagnostic commands

The domain node exposes a *Public API* that is used by participant nodes to communicate with the synchronization domain. This must be accessible from where the participant nodes are hosted.

Similar to the participant node, a domain node also exposes an Admin API for administration services. You can use these to manage keys, set domain parameters and enable or disable participant nodes within a domain, for example. The console provides access to the Admin APIs of the configured participants and domains.

Note: Canton's Admin APIs must not be confused with the `admin` package of the Ledger API. The `admin` package of the Ledger API provides services for managing parties and packages on *any Daml participant*. Canton's Admin APIs allows you to administrate *Canton-based nodes*. Both the `participant` and the `domain` nodes expose an Admin API with partially overlapping functionality.

Furthermore, participant and domain nodes communicate with each other through the Public API. The participants do not communicate with each other directly, but are free to connect to as many domains as they desire.

As you can see, nothing in the configuration specifies that our `participant1` and `participant2` should connect to `mydomain`. Canton connections are not statically configured – they are added dynamically. So first, let's connect the participants to the domain.

1.23.4.4 Connecting The Nodes

Using the console we can run commands on each of the configured (participant or domain) nodes. As such, we can check the health of a node using the *health.status* command:

```
@ health.status
res5: EnterpriseCantonStatus = Status for Domain 'mydomain':
Domain id:
 ↪mydomain::1220b4e9b0f09429d18bb4f197864468b713b28d5334e7581e82e6b9f129cf5c0e15
Uptime: 7.494604s
Ports:
    admin: 30103
    public: 30102
Connected Participants: None
Sequencer: SequencerHealthStatus(active = true)
Components:
    sequencer : Ok()
    memory_storage : Ok()
    domain-topology-sender : Ok()

Status for Participant 'participant1':
Participant id:
 ↪PAR::participant1::1220e92602e979f678f3b64664f2599a03ebccdd3e914d24c3695d7e4bcfdc77734a
Uptime: 5.181514s
Ports:
    ledger: 30098
    admin: 30099
Connected domains: None
```

(continues on next page)

(continued from previous page)

```
Unhealthy domains: None
Active: true
Components:
    memory_storage : Ok()
    sync-domain : Not Initialized
    sync-domain-ephemeral : Not Initialized
    sequencer-client : Not Initialized

Status for Participant 'participant2':
Participant id:
 ↪PAR::participant2::12207f6b1097871943e7f365a3f57d388d635561284143441aed3d1abda119c7b57e
Uptime: 3.406213s
Ports:
    ledger: 30100
    admin: 30101
Connected domains: None
Unhealthy domains: None
Active: true
Components:
    memory_storage : Ok()
    sync-domain : Not Initialized
    sync-domain-ephemeral : Not Initialized
    sequencer-client : Not Initialized
```

We can do this also individually on each node. As an example, to query the status of `participant1`:

```
@ participant1.health.status
res6: com.digitalasset.canton.health.admin.data.NodeStatus[com.digitalasset.
 ↪canton.health.admin.data.ParticipantStatus] = Participant id:
 ↪PAR::participant1::1220e92602e979f678f3b64664f2599a03ebccdd3e914d24c3695d7e4bcfdc77734a
Uptime: 5.349261s
Ports:
    ledger: 30098
    admin: 30099
Connected domains: None
Unhealthy domains: None
Active: true
Components:
    memory_storage : Ok()
    sync-domain : Not Initialized
    sync-domain-ephemeral : Not Initialized
    sequencer-client : Not Initialized
```

or for the domain:

```
@ mydomain.health.status
res7: com.digitalasset.canton.health.admin.data.NodeStatus[mydomain.Status] =
 ↪Domain id:
 ↪mydomain::1220b4e9b0f09429d18bb4f197864468b713b28d5334e7581e82e6b9f129cf5c0e15
Uptime: 7.854944s
Ports:
    admin: 30103
    public: 30102
Connected Participants: None
Sequencer: SequencerHealthStatus(active = true)
```

(continues on next page)

(continued from previous page)

```
Components:
    sequencer : Ok()
    memory_storage : Ok()
    domain-topology-sender : Ok()
```

Recall that the aliases `mydomain`, `participant1` and `participant2` come from the configuration file. By default, Canton will start and initialize the nodes automatically. This behavior can be overridden using the `--manual-start` command line flag or appropriate configuration settings.

For the moment, ignore the long hexadecimal strings that follow the node aliases; these have to do with Canton's identities, which we will explain shortly. As you see, the domain doesn't have any connected participants, and the participants are also not connected to any domains.

To connect the participants to the domain:

```
@ participant1.domains.connect_local(mydomain)
```

```
@ participant2.domains.connect_local(mydomain)
```

Now, check the status again:

```
@ health.status
res10: EnterpriseCantonStatus = Status for Domain 'mydomain':
Domain id:
 ↪mydomain::1220b4e9b0f09429d18bb4f197864468b713b28d5334e7581e82e6b9f129cf5c0e15
Uptime: 10.87264s
Ports:
    admin: 30103
    public: 30102
Connected Participants:
    PAR::participant1::1220e92602e9...
    PAR::participant2::12207f6b1097...
Sequencer: SequencerHealthStatus(active = true)
Components:
    sequencer : Ok()
    memory_storage : Ok()
    domain-topology-sender : Ok()

..
```

As you can read from the status, both participants are now connected to the domain. You can test the connection with the following diagnostic command, inspired by the ICMP ping:

```
@ participant1.health.ping(participant2)
res11: Duration = 613 milliseconds
```

If everything is set up correctly, this will report the roundtrip time between the Ledger APIs of the two participants. On the first attempt, this time will probably be several seconds, as the JVM is warming up. This will decrease significantly on the next attempt, and decrease again after JVM's just-in-time compilation kicks in (by default this is after 10000 iterations).

You have just executed your first smart contract transaction over Canton. Every participant node has an associated built-in party that can take part in smart contract interactions. The `ping` command uses a particular smart contract that is by default pre-installed on every Canton participant. In fact,

the command uses the Admin API to access a pre-installed application, which then issues Ledger API commands operating on this smart contract.

In theory, you could use your participant node's built-in party for all your application's smart contract interactions, but it's often useful to have more parties than participants. For example, you might want to run a single participant node within a company, with each employee being a separate party. For this, you need to be able to provision parties.

1.23.4.5 Canton Identities and Provisioning Parties

In Canton, the identity of each party, participant, or domain is represented by a *unique identifier*. A unique identifier consists of two components: a human-readable string and the fingerprint of a public key. When displayed in Canton the components are separated by a double colon. You can see the identifiers of the participants and the domains by running the following in the console:

```
@ mydomain.id
res12: DomainId = mydomain::1220b4e9b0f0...
```

```
@ participant1.id
res13: ParticipantId = PAR::participant1::1220e92602e9...
```

```
@ participant2.id
res14: ParticipantId = PAR::participant2::12207f6b1097...
```

The human-readable strings in these unique identifiers are derived from the local aliases by default, but can be set to any string of your choice. The public key, which is called a *namespace*, is the root of trust for this identifier. This means that in Canton, any action taken in the name of this identity must be either:

signed by this namespace key, or
signed by a key that is authorized by the namespace key to speak in the name of this identity, either directly or indirectly (e.g., if `k1` can speak in the name of `k2` and `k2` can speak in the name of `k3`, then `k1` can also speak in the name of `k3`).

In Canton, it's possible to have several unique identifiers that share the same namespace - you'll see examples of that shortly. However, if you look at the identities resulting from your last console commands, you will see that they belong to different namespaces. By default, each Canton node generates a fresh asymmetric key pair (the secret and public keys) for its own namespace when first started. The key is then stored in the storage, and reused later in case the storage is persistent (recall that `simple-topology.conf` uses memory storage, which is not persistent).

1.23.4.6 Creating Parties

You will next create two parties, Alice and Bob. Alice will be hosted at `participant1`, and her identity will use the namespace of `participant1`. Similarly, Bob will use `participant2`. Canton provides a handy macro for this:

```
@ val alice = participant1.parties.enable("Alice")
alice : PartyId = Alice::1220e92602e9...
```

```
@ val bob = participant2.parties.enable("Bob")
bob : PartyId = Bob::12207f6b1097...
```

This creates the new parties in the participants' respective namespaces. It also notifies the domain of the new parties and allows the participants to submit commands on behalf of those parties. The domain allows this since, e.g., Alice's unique identifier uses the same namespace as `participant1` and `participant1` holds the secret key of this namespace. You can check that the parties are now known to `mydomain` by running the following:

```
@ mydomain.parties.list("Alice")
res17: Seq[ListPartiesResult] = Vector(
  ListPartiesResult(
    party = Alice::1220e92602e9...,
    participants = Vector(
      ParticipantDomains(
        participant = PAR::participant1::1220e92602e9...,
        domains = Vector(
          DomainPermission(domain = mydomain::1220b4e9b0f0..., permission =
↪Submission)
        )
      )
    )
  )
)
```

and the same for Bob:

```
@ mydomain.parties.list("Bob")
res18: Seq[ListPartiesResult] = Vector(
  ListPartiesResult(
    party = Bob::12207f6b1097...,
    participants = Vector(
      ParticipantDomains(
        participant = PAR::participant2::12207f6b1097...,
        domains = Vector(
          DomainPermission(domain = mydomain::1220b4e9b0f0..., permission =
↪Submission)
        )
      )
    )
  )
)
```

1.23.4.7 Extracting Identifiers

Canton identifiers can be long strings. They are normally truncated for convenience. However, in some cases we do have to extract these identifiers so they can be shared through other channels. As an example, if you have two participants that run in completely different locations, without a shared console, then you can't ping as we did before:

```
@ participant1.health.ping(participant2)
..
```

Instead, extract the participant id of one node:

```
@ val extractedId = participant2.id.toProtoPrimitive
extractedId : String =
↪"PAR::participant2::12207f6b1097871943e7f365a3f57d388d635561284143441aed3d1abda119c7b57e
↪"
```

(continues on next page)

(continued from previous page)

This id can then be shared with the other participant, who in turn can parse the id back into an appropriate object:

```
@ val p2Id = ParticipantId.tryFromProtoPrimitive(extractedId)
p2Id : ParticipantId = PAR::participant2::12207f6b1097...
```

And subsequently, this id can be used to ping as well:

```
@ participant1.health.ping(p2Id)
res22: Duration = 576 milliseconds
```

This also works for party identifiers:

```
@ val aliceAsStr = alice.toProtoPrimitive
aliceAsStr : String =
↪"Alice::1220e92602e979f678f3b64664f2599a03ebccdd3e914d24c3695d7e4bcfdc77734a"
```

```
@ val aliceParsed = PartyId.tryFromProtoPrimitive(aliceAsStr)
aliceParsed : PartyId = Alice::1220e92602e9...
```

Generally, a Canton identity boils down to a `UniqueIdentifier` and the context in which this identifier is used. This allows you to directly access the identifier serialization:

```
@ val p2UidString = participant2.id.uid.toProtoPrimitive
p2UidString : String =
↪"participant2::12207f6b1097871943e7f365a3f57d388d635561284143441aed3d1abda119c7b57e
↪"
```

```
@ val p2FromUid = ParticipantId(UniqueIdentifier.
↪tryFromProtoPrimitive(p2UidString))
p2FromUid : ParticipantId = PAR::participant2::12207f6b1097...
```

1.23.4.8 Provisioning Smart Contract Code

To create a contract between Alice and Bob, you must first provision the contract's code to both of their hosting participants. Canton supports smart contracts written in Daml. A Daml contract's code is specified using a Daml *contract template*; an actual contract is then a *template instance*. Daml templates are packaged into *Daml archives*, or DARs for short. For this tutorial, use the pre-packaged `dars/CantonExamples.dar` file. To provision it to both `participant1` and `participant2`, you can use the `participants.all` bulk operator:

```
@ participants.all.dars.upload("dars/CantonExamples.dar")
res27: Map[com.digitalasset.canton.console.ParticipantReference, String] = Map(
  Participant 'participant1' ->
↪"1220c783022e36adf132a905711d40850477d4b817e39f1b44d62af0f4a7a3c05476",
  Participant 'participant2' ->
↪"1220c783022e36adf132a905711d40850477d4b817e39f1b44d62af0f4a7a3c05476"
)
```

The bulk operator allows you to run certain commands on a series of nodes. Canton supports the bulk operators on the generic `nodes`:

```
@ nodes.local
res28: Seq[com.digitalasset.canton.console.LocalInstanceReferenceCommon] =
↪ArraySeq(Participant 'participant1', Participant 'participant2', Domain
↪'mydomain')
```

or on the specific node type:

```
@ participants.all
res29: Seq[com.digitalasset.canton.console.ParticipantReference] =
↪List(Participant 'participant1', Participant 'participant2')
```

Allowed suffixes are `.local`, `.all` or `.remote`, where the remote refers to *remote nodes*, which we won't use here.

To validate that the DAR has been uploaded, run:

```
@ participant1.dars.list()
res30: Seq[com.digitalasset.canton.participant.admin.v0.DarDescription] = Vector(
  DarDescription(
    hash = "1220c783022e36adf132a905711d40850477d4b817e39f1b44d62af0f4a7a3c05476",
    name = "CantonExamples"
  ),
  DarDescription(
    hash = "122012a6f2b7c0b666e7541ce6f5d4273ab8d00da671b4d3bbb9bebb6a5120ec02c5",
    name = "AdminWorkflowsWithVacuuming"
  )
)
```

and on the second participant, run:

```
@ participant2.dars.list()
res31: Seq[com.digitalasset.canton.participant.admin.v0.DarDescription] = Vector(
  DarDescription(
    hash = "1220c783022e36adf132a905711d40850477d4b817e39f1b44d62af0f4a7a3c05476",
    name = "CantonExamples"
  ),
  DarDescription(
    hash = "122012a6f2b7c0b666e7541ce6f5d4273ab8d00da671b4d3bbb9bebb6a5120ec02c5",
    name = "AdminWorkflowsWithVacuuming"
  )
)
```

One important observation is that you cannot list the uploaded DARs on the domain `mydomain`. You will simply get an error if you run `mydomain.dars.list()`. This is due the fact that the domain does not know anything about Daml or smart contracts. All the contract code is only executed by the involved participants on a need to know basis and needs to be explicitly enabled by them.

Now you are ready to actually start running smart contracts using Canton.

1.23.4.9 Executing Smart Contracts

Let's start by looking at some smart contract code. In our example, we'll have three parties, Alice, Bob and the Bank. In the scenario, Alice and Bob will agree that Bob has to paint her house. In exchange, Bob will get a digital bank note (I-Owe-You, IOU) from Alice, issued by a bank.

First, we need to add the Bank as a party:

```
@ val bank = participant2.parties.enable("Bank", waitForDomain = DomainChoice.All)
bank : PartyId = Bank::12207f6b1097...
```

You might have noticed that we've added a `waitForDomain` argument here. This is necessary to force some synchronisation between the nodes to ensure that the new party is known within the distributed system before it is used.

Note: Canton alleviates most synchronization issues when interacting with Daml contracts. Nevertheless, Canton is a concurrent, distributed system. All operations happen asynchronously. Creating the `Bank` party is an operation local to `participant2`, and `mydomain` becomes aware of the party with a delay (see *Topology Transactions* for more detail). Processing and network delays also exist for all other operations that affect multiple nodes, though everyone sees the operations on the domain in the same order. When you execute commands interactively, the delays are usually too small to notice. However, if you're programming Canton scripts or applications that talk to multiple nodes, you might need some form of manual synchronization. Most Canton console commands have some form of synchronisation to simplify your life and sometimes, using `utils.retry_until_true(...)` is a handy solution.

The corresponding Daml contracts that we are going to use for this example are:

```
module Iou where

import Daml.Script

data Amount = Amount {value: Decimal; currency: Text} deriving (Eq, Ord, Show)

amountAsText (amount : Amount) : Text = show amount.value <> amount.currency

template Iou
  with
    payer: Party
    owner: Party
    amount: Amount
    viewers: [Party]
  where

    ensure (amount.value >= 0.0)

    signatory payer
    observer owner
    observer viewers

    choice Call : ContractId GetCash
      controller owner
      do
        create GetCash with payer; owner; amount
```

(continues on next page)

(continued from previous page)

```
    choice Transfer : ContractId Iou
      with
        newOwner: Party
      controller owner
      do
        create this with owner = newOwner; viewers = []

    choice Share : ContractId Iou
      with
        viewer : Party
      controller owner
        do
          create this with viewers = (viewer :: viewers)
```

```
module Paint where

import Daml.Script
import Iou

template PaintHouse
  with
    painter: Party
    houseOwner: Party
  where
    signatory painter, houseOwner
    agreement
      show painter <> " will paint the house of " <> show houseOwner

template OfferToPaintHouseByPainter
  with
    houseOwner: Party
    painter: Party
    bank: Party
    amount: Amount
  where
    signatory painter
    observer houseOwner

    choice AcceptByOwner : ContractId Iou
      with
        iouId : ContractId Iou
      controller houseOwner
      do
        iouId2 <- exercise iouId Transfer with newOwner = painter
        paint <- create $ PaintHouse with painter; houseOwner
        return iouId2
```

We won't dive into the details of Daml, as this is explained elsewhere. But one key observation is that the contracts themselves are passive. The contract instances represent the ledger and only encode the rules according to which the ledger state can be changed. Any change requires you to trigger some Daml contract execution by sending the appropriate commands over the Ledger API.

The Canton console gives you interactive access to this API, together with some utilities that can be useful for experimentation. The Ledger API uses gRPC.

In theory, we would need to compile the Daml code into a DAR and then upload it to the participant nodes. We actually did this already by uploading the `CantonExamples.dar`, which includes the contracts. Now we can create our first contract using the template `Iou.Iou`. The name of the template is not enough to uniquely identify it. We also need the package id, which is just the *sha256* hash of the binary module containing the respective template.

Find that package by running:

```
@ val pkgIou = participant1.packages.find("Iou").head
pkgIou : com.digitalasset.canton.participant.admin.v0.PackageDescription =
 ↪PackageDescription(
  packageId = "9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0",
  sourceDescription = "CantonExamples"
)
```

Using this package-id, we can create the IOU:

```
@ val createIouCmd = ledger_api_utils.create(pkgIou.packageId,"Iou","Iou",Map(
 ↪"payer" -> bank,"owner" -> alice,"amount" -> Map("value" -> 100.0, "currency" ->
 ↪ "EUR"),"viewers" -> List()))
createIouCmd : com.daml.ledger.api.v1.commands.Command = Command(
  command = Create(
    value = CreateCommand(
      templateId = Some(
        value = Identifier(
          packageId =
 ↪"9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0",
..
```

and then send that command to the Ledger API:

```
@ participant2.ledger_api.commands.submit(Seq(bank), Seq(createIouCmd))
res35: com.daml.ledger.api.v1.transaction.TransactionTree = TransactionTree(
  transactionId =
 ↪"122016dfb107997decae572917a0cca323f3d71a99d808c55c821fc84921bee57bbc",
  commandId = "66d0c0bd-5a2f-4a46-a933-31dbb65bb856",
  workflowId = "",
  effectiveAt = Some(
    value = Timestamp(
      seconds = 1686572349L,
      nanos = 575851000,
      unknownFields = UnknownFieldSet(fields = Map())
    )
  ),
  offset = "00000000000000015",
..
```

Here, we've submitted this command as party *Bank* on participant2. Interestingly, we can test here the Daml authorization logic. As the *signatory* of the contract is *Bank*, we can't have Alice submitting the contract:

```
@ participant1.ledger_api.commands.submit(Seq(alice), Seq(createIouCmd))
ERROR com.digitalasset.canton.integration.EnterpriseEnvironmentDefinition$$anon$3
 ↪- Request failed for participant1.
  GrpcClientError: INVALID_ARGUMENT/DAML_AUTHORIZATION_ERROR(8,9d7c7884):
 ↪Interpretation error: Error: node NodeId(0)
 ↪(9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0:Iou:Iou)
```

(continues on next page)

```
 ↪requires authorizers
 ↪Bank::12207f6b1097871943e7f365a3f57d388d635561284143441aed3d1abda119c7b57e, but
 ↪only
 ↪Alice::1220e92602e979f678f3b64664f2599a03ebccdd3e914d24c3695d7e4bcfdc77734a
 ↪were given
```

(continued from previous page)

```
  Request: SubmitAndWaitTransactionTree(actAs = Alice::1220e92602e9..., readAs =
↪Seq(), commandId = '', workflowId = '', submissionId = '', deduplicationPeriod
↪= None(), commands = ...)
  CorrelationId: 9d7c7884-6e76-41c2-b70b-665c49bd097f
..
```

And Alice cannot impersonate the Bank by pretending to be it (on her participant):

```
@ participant1.ledger_api.commands.submit(Seq(bank), Seq(createIouCmd))
ERROR com.digitalasset.canton.integration.EnterpriseEnvironmentDefinition$$anon$3
↪- Request failed for participant1.
  GrpcRequestRefusedByServer: NOT_FOUND/NO_DOMAIN_ON_WHICH_ALL_SUBMITTERS_CAN_
↪SUBMIT(11,89c4dc65): This participant can not submit as the given submitter on
↪any connected domain
  Request: SubmitAndWaitTransactionTree(actAs = Bank::12207f6b1097..., readAs =
↪Seq(), commandId = '', workflowId = '', submissionId = '', deduplicationPeriod
↪= None(), commands = ...)
  CorrelationId: 89c4dc654bf60571a516aa17b36abeb8
..
```

Alice can, however, observe the contract on her participant by searching her *Active Contract Set* (ACS) for it:

```
@ val aliceIou = participant1.ledger_api.acs.find_generic(alice, _.templateId.
↪isModuleEntity("Iou", "Iou"))
aliceIou : com.digitalasset.canton.admin.api.client.commands.
↪LedgerApiTypeWrappers.WrappedCreatedEvent = WrappedCreatedEvent(
  event = CreatedEvent(
    eventId = "
↪#122016dfb107997decae572917a0cca323f3d71a99d808c55c821fc84921bee57bbc:0",
    contractId =
↪"0064eace0d06c962a4141372442e1b64b4655383df07f1ea191a90094ed3df35dcca01122098f4a7f6a3945b
↪",
..
```

We can check Alice's ACS, which will show us all the contracts Alice knows about:

```
@ participant1.ledger_api.acs.of_party(alice)
res37: Seq[com.digitalasset.canton.admin.api.client.commands.
↪LedgerApiTypeWrappers.WrappedCreatedEvent] = List(
  WrappedCreatedEvent(
    event = CreatedEvent(
      eventId = "
↪#122016dfb107997decae572917a0cca323f3d71a99d808c55c821fc84921bee57bbc:0",
      contractId =
↪"0064eace0d06c962a4141372442e1b64b4655383df07f1ea191a90094ed3df35dcca01122098f4a7f6a3945b
↪",
      templateId = Some(
        value = Identifier(
          packageId =
↪"9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0",
..
```

As expected, Alice does see exactly the contract that the Bank previously created. The command returns a sequence of wrapped CreatedEvent's. This Ledger API data type represents the event of a contract's creation. The output is a bit verbose, but the wrapper provides convenient functions to

manipulate the `CreatedEvent`s in the Canton console:

```
@ participant1.ledger_api.acs.of_party(alice).map(x => (x.templateId, x.
↪arguments))
res38: Seq[(TemplateId, Map[String, Any])] = List(
  (
    TemplateId(
      packageId =
↪"9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0",
      moduleName = "Iou",
      entityName = "Iou"
    ),
    HashMap(
      "payer" ->
↪"Bank::12207f6b1097871943e7f365a3f57d388d635561284143441aed3d1abda119c7b57e",
      "viewers" -> List(elements = Vector()),
      "owner" ->
↪"Alice::1220e92602e979f678f3b64664f2599a03ebccdd3e914d24c3695d7e4bcfdc77734a",
      "amount.currency" -> "EUR",
      "amount.value" -> "100.0000000000"
    )
  )
)
```

Going back to our story, Bob now wants to offer to paint Alice's house in exchange for money. Again, we need to grab the package id, as the Paint contract is in a different module:

```
@ val pkgPaint = participant1.packages.find("Paint").head
pkgPaint : com.digitalasset.canton.participant.admin.v0.PackageDescription =
↪PackageDescription(
  packageId = "9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0",
  sourceDescription = "CantonExamples"
)
```

Note that the modules are compositional. The `Iou` module is not aware of the `Paint` module, but the `Paint` module is using the `Iou` module within its workflow. This is how we can extend any workflow in Daml and build on top of it. In particular, the Bank does not need to know about the `Paint` module at all, but can still participate in the transaction without any adverse effect. As a result, everybody can extend the system with their own functionality. Let's create and submit the offer now:

```
@ val createOfferCmd = ledger_api_utils.create(pkgPaint.packageId, "Paint",
↪"OfferToPaintHouseByPainter", Map("bank" -> bank, "houseOwner" -> alice,
↪"painter" -> bob, "amount" -> Map("value" -> 100.0, "currency" -> "EUR")))
createOfferCmd : com.daml.ledger.api.v1.commands.Command = Command(
  command = Create(
    value = CreateCommand(
      templateId = Some(
        value = Identifier(
          packageId =
↪"9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0",
..
```

```
@ participant2.ledger_api.commands.submit_flat(Seq(bob), Seq(createOfferCmd))
res41: com.daml.ledger.api.v1.transaction.Transaction = Transaction(
  transactionId =
↪"1220ab0b25094769ffd759d1e4c33fd2924212abe93c4f1f997ce3e619643ec63d42",
```

(continues on next page)

(continued from previous page)

```
  commandId = "b63e26ee-7c29-4ea7-849c-79796a7b5e5b",
  workflowId = "",
  effectiveAt = Some(
    value = Timestamp(
..
```

Alice will observe this offer on her node:

```
@ val paintOffer = participant1.ledger_api.acs.find_generic(alice, _.templateId.
↪isModuleEntity("Paint", "OfferToPaintHouseByPainter"))
paintOffer : com.digitalasset.canton.admin.api.client.commands.
↪LedgerApiTypeWrappers.WrappedCreatedEvent = WrappedCreatedEvent(
  event = CreatedEvent(
    eventId = "
↪#1220ab0b25094769ffd759d1e4c33fd2924212abe93c4f1f997ce3e619643ec63d42:0",
    contractId =
↪"0021ae8b91a08ed8f073d0331cb370b6ce0f61417478731ca6a4488cb248f21ba6ca01122016b5004bb68ae2
↪",
    templateId = Some(
      value = Identifier(
..
```

1.23.4.10 Privacy

Looking at the ACS of Alice, Bob and the Bank, we note that Bob sees only the paint offer:

```
@ participant2.ledger_api.acs.of_party(bob).map(x => (x.templateId, x.arguments))
res43: Seq[(TemplateId, Map[String, Any])] = List(
  (
    TemplateId(
      packageId =
↪"9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0",
      moduleName = "Paint",
      entityName = "OfferToPaintHouseByPainter"
    ),
    HashMap(
      "painter" ->
↪"Bob::12207f6b1097871943e7f365a3f57d388d635561284143441aed3d1abda119c7b57e",
      "houseOwner" ->
↪"Alice::1220e92602e979f678f3b64664f2599a03ebccdd3e914d24c3695d7e4bcfdc77734a",
      "bank" ->
↪"Bank::12207f6b1097871943e7f365a3f57d388d635561284143441aed3d1abda119c7b57e",
      "amount.currency" -> "EUR",
      "amount.value" -> "100.0000000000"
    )
  )
)
```

while the Bank sees the Iou contract:

```
@ participant2.ledger_api.acs.of_party(bank).map(x => (x.templateId, x.arguments))
res44: Seq[(TemplateId, Map[String, Any])] = List(
  (
    TemplateId(
```

(continues on next page)

(continued from previous page)

```
      packageId =
↪"9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0",
      moduleName = "Iou",
      entityName = "Iou"
    ),
    HashMap(
      "payer" ->
↪"Bank::12207f6b1097871943e7f365a3f57d388d635561284143441aed3d1abda119c7b57e",
      "viewers" -> List(elements = Vector()),
      "owner" ->
↪"Alice::1220e92602e979f678f3b64664f2599a03ebccdd3e914d24c3695d7e4bcfdc77734a",
      "amount.currency" -> "EUR",
      "amount.value" -> "100.0000000000"
    )
  )
)
```

But Alice sees both on her participant node:

```
@ participant1.ledger_api.acs.of_party(alice).map(x => (x.templateId, x.
↪arguments))
res45: Seq[(TemplateId, Map[String, Any])] = List(
  (
    TemplateId(
      packageId =
↪"9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0",
      moduleName = "Iou",
      entityName = "Iou"
    ),
    HashMap(
      "payer" ->
↪"Bank::12207f6b1097871943e7f365a3f57d388d635561284143441aed3d1abda119c7b57e",
      "viewers" -> List(elements = Vector()),
      "owner" ->
↪"Alice::1220e92602e979f678f3b64664f2599a03ebccdd3e914d24c3695d7e4bcfdc77734a",
      "amount.currency" -> "EUR",
      "amount.value" -> "100.0000000000"
    )
  ),
  (
    TemplateId(
      packageId =
↪"9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0",
      moduleName = "Paint",
      entityName = "OfferToPaintHouseByPainter"
    ),
    HashMap(
      "painter" ->
↪"Bob::12207f6b1097871943e7f365a3f57d388d635561284143441aed3d1abda119c7b57e",
      "houseOwner" ->
↪"Alice::1220e92602e979f678f3b64664f2599a03ebccdd3e914d24c3695d7e4bcfdc77734a",
      "bank" ->
↪"Bank::12207f6b1097871943e7f365a3f57d388d635561284143441aed3d1abda119c7b57e",
      "amount.currency" -> "EUR",
      "amount.value" -> "100.0000000000"
    )
```

(continues on next page)

(continued from previous page)

```
  )
)
```

If there were a third participant node, it wouldn't have even noticed that there was anything happening, let alone have received any contract data. Or if we had deployed the Bank on that third node, that node would not have been informed about the Paint offer. This privacy feature goes so far in Canton that not even everybody within a single atomic transaction is aware of each other. This is a property unique to the Canton synchronization protocol, which we call *sub-transaction privacy*. The protocol ensures that only eligible participants will receive any data. Furthermore, while the node running `mydomain` does receive this data, the data is encrypted and `mydomain` cannot read it.

We can run such a step with sub-transaction privacy by accepting the offer, which will lead to the transfer of the Bank Iou, without the Bank actually learning about the Paint agreement:

```
@ import com.digitalasset.canton.protocol.LfContractId
```

```
@ val acceptOffer = ledger_api_utils.exercise("AcceptByOwner", Map("iouId" ->
↪LfContractId.assertFromString(aliceIou.event.contractId)),paintOffer.event)
acceptOffer : com.daml.ledger.api.v1.commands.Command = Command(
  command = Exercise(
    value = ExerciseCommand(
      templateId = Some(
        value = Identifier(
          packageId =
↪"9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0",
..
```

```
@ participant1.ledger_api.commands.submit_flat(Seq(alice), Seq(acceptOffer))
res48: com.daml.ledger.api.v1.transaction.Transaction = Transaction(
  transactionId =
↪"1220ced37b240eefc96341fa42245989e4750f8b777121d445dfb3d2688ee625c08c",
  commandId = "7b231bb3-aa08-4f6e-a4e4-d34160e893d0",
  workflowId = "",
  effectiveAt = Some(
    value = Timestamp(
..
```

Note that the conversion to `LfContractId` was required to pass in the Iou contract id as the correct type.

1.23.4.11 Your Development Choices

While the `ledger_api` functions in the Console can be handy for educational purposes, the Daml SDK provides you with much more convenient tools to inspect and manipulate the ledger content:

- The browser based Navigator
- The console version Navigator
- Daml script for scripting
- Daml triggers for reactive operations
- Daml REPL for interactive manipulations
- Json API for browser based UIs
- Bindings in a variety of languages to build your own applications

All these tools work against the Ledger API.

1.23.4.12 Automation using bootstrap scripts

You can configure a bootstrap script to avoid having to manually complete routine tasks such as starting nodes or provisioning parties each time Canton is started. Bootstrap scripts are automatically run after Canton has started and can contain any valid Canton Console commands. A bootstrap script is passed via the `--bootstrap` CLI argument when starting Canton. By convention, we use a `.canton` file ending.

For example, the bootstrap script to connect the participant nodes to the local domain and ping participant1 from participant2 (see *Starting and Connecting The Nodes*) is:

```
// start all local instances defined in the configuration file
nodes.local.start()

// Connect participant1 to mydomain using the connect macro.
// The connect macro will inspect the domain configuration to find the correct
 ↪URL and Port.
// The macro is convenient for local testing, but obviously doesn't work in a
 ↪distributed setup.
participant1.domains.connect_local(mydomain)

val mydomainPort = Option(System.getProperty("canton-examples.mydomain-port")).
 ↪getOrElse("5018")

// Connect participant2 to mydomain using just the target URL and a local name we
 ↪use to refer to this particular
// connection. This is actually everything Canton requires and this second type
 ↪of connect call can be used
// in order to connect to a remote Canton domain.
//
// The connect call is just a wrapper that invokes the `domains.register`,
 ↪`domains.get_agreement` and `domains.accept_agreement` calls.
//
// The address can be either HTTP or HTTPS. From a security perspective, we do
 ↪assume that we either trust TLS to
// initially introduce the domain. If we don't trust TLS for that, we can also
 ↪optionally include a so called
// EssentialState that establishes the trust of the participant to the domain.
// Whether a domain will let a participant connect or not is at the discretion of
 ↪the domain and can be configured
// there. While Canton establishes the connection, we perform a handshake,
 ↪exchanging keys, authorizing the connection
// and verifying version compatibility.
participant2.domains.connect("mydomain", s"http://localhost:$mydomainPort")

// The above connect operation is asynchronous. It is generally at the discretion
 ↪of the domain
// to decide if a participant can join and when. Therefore, we need to
 ↪asynchronously wait here
// until the participant observes its activation on the domain. As the domain is
 ↪configured to be
// permissionless in this example, the approval will be granted immediately.
utils.retry_until_true {
```

(continues on next page)

(continued from previous page)

```
    participant2.domains.active("mydomain")
}

participant2.health.ping(participant1)
```

Note how we again use `retry_until_true` to add a manual synchronization point, making sure that participant2 is registered, before proceeding to ping participant1.

1.23.4.13 What Next?

You are now ready to start using Canton for serious tasks. If you want to develop a Daml application and run it on Canton, we recommend the following resources:

1. Install the Daml SDK to get access to the Daml IDE and other tools, such as the Navigator.
2. Run through the *Daml SDK getting-started example* to learn how to build your own Daml applications on Canton.
3. Follow the Daml documentation to learn how to program new contracts, or check out the Daml Examples to find existing ones for your needs.
4. Use the Navigator for easy Web-based access and manipulation of your contracts.

If you want to understand more about Canton:

1. Read the *requirements* that Canton was built for to find out more about the properties of Canton.
2. Read the *architectural overview* for more understanding of Canton concepts and internals.

If you want to deploy your own Canton nodes, consult the *installation guide*.

1.23.5 Daml SDK and Canton

This tutorial shows how to run an application on a distributed setup using Canton instead of running it on the Daml sandbox. This comes with a few known problems and this section explains how to work around them.

In this tutorial, you will learn how to run the Create Daml App example on Canton. This guide will teach you:

1. The main concepts of Daml
2. How to compile your own Daml Archive (DAR)
3. How to run the Create Daml App example on Canton
4. How to write your own Daml code
5. How to integrate a conventional application with Canton

If you haven't yet done so, please run through the *Getting Started with Canton* and the original Daml getting started guide to familiarise yourself with the example application. Then come back here to get the same example running on Canton.

1.23.5.1 Starting Canton

Follow the Daml SDK installation guide to get the SDK locally installed.

This guide has been tested with the SDK version 2.5.1. Set the environment variable `DAML_SDK_VERSION` to `2.5.1` so that subsequent `daml` commands use this version.

```
export DAML_SDK_VERSION=2.5.1
```

Starting from the location where you unpacked the Canton distribution, fetch the create-daml-app example into a directory named `create-daml-app` (as the example configuration files of `examples/04-create-daml-app` expect the files to be there):

```
daml new create-daml-app --template create-daml-app
```

Next, compile the Daml code into a DAR file (this will create the file `.daml/dist/create-daml-app-0.1.0.dar`), and run the code generation step used by the UI:

```
cd create-daml-app
daml build
daml codegen js .daml/dist/create-daml-app-0.1.0.dar -o ui/daml.js
```

You will also need to install the dependencies for the UI:

```
cd ui
npm install
```

Next, the original tutorial would ask you to start the Sandbox and the HTTP JSON API with `daml start`. We will instead start Canton using the distributed setup in `examples/04-create-daml-app`, and will later start the HTTP JSON API in a separate step.

Return to the directory where you unpacked the Canton distribution and start Canton with:

```
cd ../..
bin/canton -c examples/04-create-daml-app/canton.conf --bootstrap examples/04-
↪create-daml-app/init.canton
```

Note: If you get an `Compilation Failed` error, you may have to make the Canton binary executable with `chmod +x bin/canton`

This will start two participant nodes, allocate the parties Alice, Bob and Public and create corresponding users `alice` and `bob`. Each participant node will expose its own ledger API:

1. Alice will be hosted by participant1, with its ledger API on port `12011`
2. Bob will be hosted by participant2, with its ledger API on port `12021`

Note that the `examples/04-create-daml-app/init.canton` script performs a few setup steps to permission the parties and upload the DAR.

Leave Canton running and switch to a new terminal window.

1.23.5.2 Running the Create Daml App Example

Once Canton is running, start the HTTP JSON API:

> Connected to the ledger API on port `12011` (corresponding to Alice's participant)
> And connected to the UI on the default expected port `7575`

```
DAML_SDK_VERSION=2.5.1 daml json-api \
    --ledger-host localhost \
    --ledger-port 12011 \
    --http-port 7575 \
    --allow-insecure-tokens
```

Leave this running. The UI can then be started from a third terminal window with:

```
cd create-daml-app/ui
REACT_APP_LEDGER_ID=participant1 npm start
```

Note that we have to configure the ledger ID used by the UI to match the name of the participant that we're running against. This is done using the environment variable `REACT_APP_LEDGER_ID`.

We can now log in as `alice`.

Connecting to participant2

You can log in as Bob using `participant2` by following essentially the same process as for participant1, adjusting the ports to correspond to `participant2`.

First, start another instance of the HTTP JSON API, this time using the options `--ledger-port=12021` and `--http-port 7576`. `12021` corresponds to participant2's ledger port, and `7576` is a new port for another instance of the HTTP JSON API:

```
DAML_SDK_VERSION=2.5.1 daml json-api \
   --ledger-host localhost \
   --ledger-port 12021 \
   --http-port 7576 \
   --allow-insecure-tokens
```

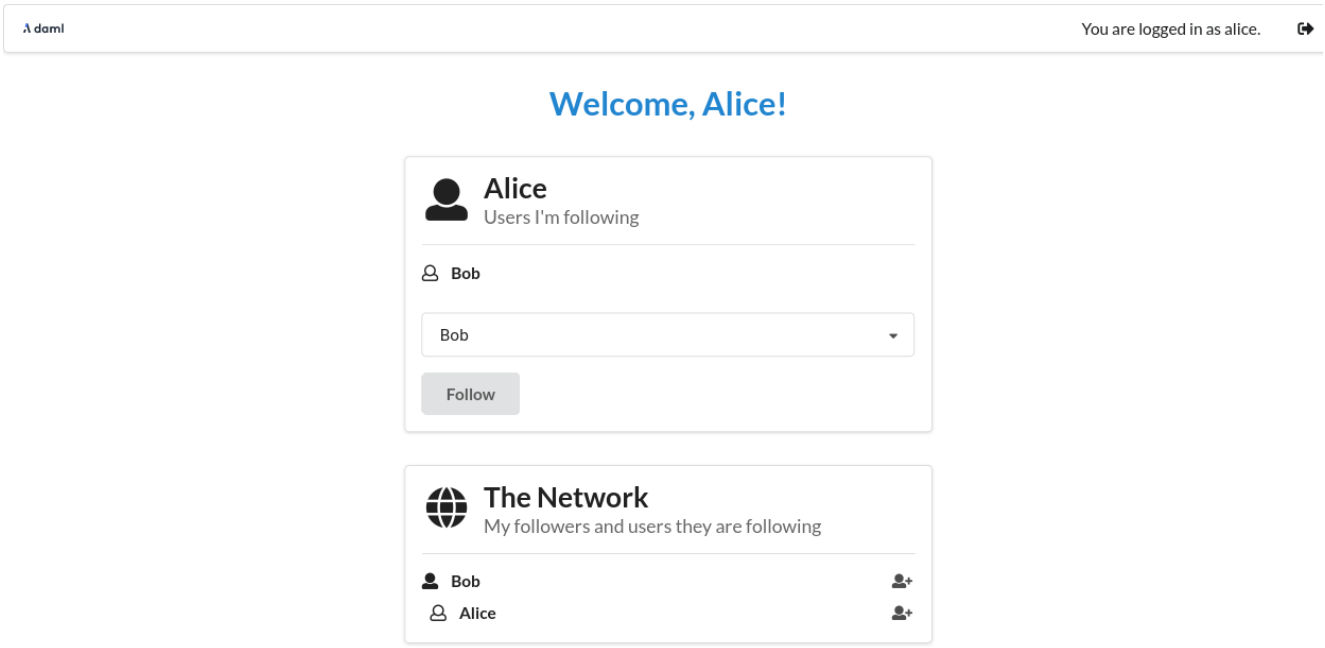
Then start another instance of the UI for Bob, running on port `3001` and connected to the HTTP JSON API on port `7576`:

```
cd create-daml-app/ui
PORT=3001 REACT_APP_HTTP_JSON=http://localhost:7576 REACT_APP_LEDGER_
↪ID=participant2 npm start
```

You can then log in with the user id `bob`.

Now that both parties have logged in, you can select `Bob` in the dropdown from Alice's view and follow him and the other way around.

After both parties have followed each other, the resulting view from Alice's side will look as follows.



Note that create-daml-app sets up human-readable aliases for party ids, which is why we can use those names to follow other parties instead of their party id.

1.23.5.3 What Next?

Now that you have started to become familiar with Daml and what a full Daml-based solution looks like, you can build your own first Daml application.

1. Use the Daml language reference docs to master Daml and build your own Daml model.
2. Test your model using Daml scripts.
3. Create a simple UI following the example of the Create Daml App template used in this tutorial.
4. See how to compose *workflows across multiple Canton domains*.
5. Showcase your application on the forum.

Composability is currently an Early Access Feature in Alpha status.

Note: The example in this tutorial uses unsupported Scala bindings and codegen.

1.23.6 Composability

In this tutorial, you will learn how to build workflows that span several Canton domains. Composability turns those several Canton domains into one conceptual ledger at the application level.

The tutorial assumes the following prerequisites:

> You have worked through the *Getting started* tutorial and know how to interact with the Canton console.
> You know the Daml concepts that are covered in the Daml introduction.
> The running example uses the *ledger API*, the Scala codegen (no longer supported by Daml) for Daml, and Canton's *identity management*. If you want to understand the example code in full, please refer to the above documentation.

The tutorial consists of two parts:

1. The *first part* illustrates how to design a workflow that spans multiple domains.
2. The *second part* shows how to compose existing workflows on different domains into a single workflow and the benefits this brings.

The Daml models are shipped with the Canton release in the `daml/CantonExamples` folder in the modules `Iou` and `Paint`. The configuration and the steps are available in the `examples/05-composability` folder of the Canton release. To run the workflow, start Canton from the release's root folder as follows:

```
./bin/canton -c examples/05-composability/composability.conf
```

You can copy-paste the console commands from the tutorial in the given order into the Canton console to run them interactively. All console commands are also summarized in the bootstrap scripts `composability1.canton`, `composability-auto-transfer.canton`, and `composability2.canton`.

Note: Note that to use composability, we do have to turn off contract key uniqueness, as uniqueness cannot be provided across multiple domains. Therefore, composability is just a preview feature and explained here to demonstrate an early version of it that is not yet suitable for production use.

1.23.6.1 Part 1: A multi-domain workflow

We consider the *paint agreement scenario* from the *Getting started* tutorial. The house owner and the painter want to enter a paint agreement that obliges the painter to paint the house owner's house. To enter such an agreement, the house owner proposes a paint offer to the painter and the painter accepts. Upon acceptance, the paint agreement shall be created atomically with changing the ownership of the money, which we represent by an IOU backed by the bank.

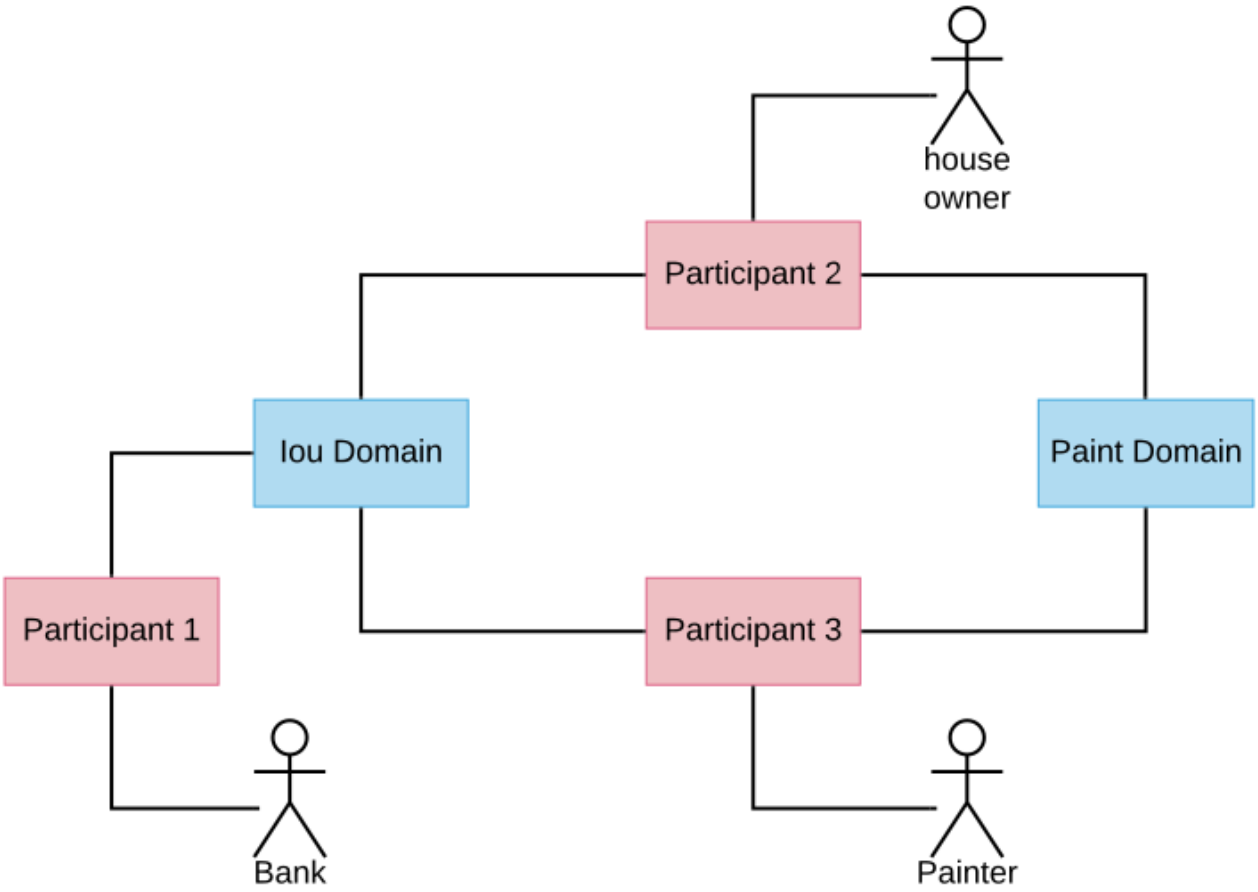
Atomicity guarantees that no party can scam the other: The painter enters the obligation of painting the house only if house owner pays, and the house owner pays only if the painter enters the obligation. This avoid bad scenarios such as the following, which would have to be resolved out of band, e.g., using legal processes:

> The house owner spends the IOU on something else and does not pay the painter, even though the painter has entered the obligation to paint the house. The painter then needs to convince the house owner to pay with another IOU or to revoke the paint agreement.
> The house owner wires the money to the painter, but the painter refuses to enter the paint agreement. The house owner then begs the painter to return the money.

Setting up the topology

In this example, we assume a topology with two domains, `iou` and `paint`. The house owner's and the painter's participants are connected to both domains, as illustrated in the following diagram.

The configuration file `composability.conf` configures the two domains `iou` and `paint` and three participants.


house
owner
Participant 2
Iou Domain
Paint Domain
Participant 1
Participant 3
Bank
Painter

```
canton {
  features {
    enable-preview-commands = yes
    enable-testing-commands = yes
  }
  monitoring {
    tracing.propagation = enabled
    logging.api.message-payloads = true
  }
  domains {
    iou {
      public-api.port = 13018
      admin-api.port = 13019
      storage.type = memory
      init.domain-parameters.unique-contract-keys = false
    }

    paint {
      public-api.port = 13028
      admin-api.port = 13029
      storage.type = memory
      init.domain-parameters.unique-contract-keys = false
    }
  }

  participants {
    participant1 {
      ledger-api.port = 13011
      admin-api.port = 13012
      storage.type = memory
      init.parameters.unique-contract-keys = false
    }

    participant2 {
      ledger-api.port = 13021
      admin-api.port = 13022
      storage.type = memory
      init.parameters.unique-contract-keys = false
    }

    participant3 {
      ledger-api.port = 13031
      admin-api.port = 13032
      storage.type = memory
      init.parameters.unique-contract-keys = false
    }
  }
}
```

As the first step, some domain parameters are changed (setting `transfer-exclusivity-timeout` will be explained in the *second part* of this tutorial). Then, all the nodes are started and the parties for the bank (hosted on participant 1), the house owner (hosted on participant 2), and the painter (hosted on participant 3) are created. The details of the party onboarding are not relevant for show-casing cross-domain workflows.

```
// update parameters
iou.service.update_dynamic_domain_parameters(
  _.update(transferExclusivityTimeout = Duration.Zero)
) // disable automatic transfer-in

paint.service.update_dynamic_domain_parameters(
  _.update(transferExclusivityTimeout = 2.seconds)
)

// connect participants to the domain
participant1.domains.connect_local(iou)
participant2.domains.connect_local(iou)
participant3.domains.connect_local(iou)
participant2.domains.connect_local(paint)
participant3.domains.connect_local(paint)

// the connect call will use the configured domain name as an alias. the
↪configured
// name is the one used in the configuration file.
// in reality, all participants pick the alias names they want, which means that
// aliases are not unique, whereas a `DomainId` is. However, the
// alias is convenient, while the DomainId is a rather long string including a
↪hash.
// therefore, for commands, we prefer to use a short alias instead.
val paintAlias = paint.name
val iouAlias = iou.name

// create the parties
val Bank = participant1.parties.enable("Bank")
val HouseOwner = participant2.parties.enable("House Owner")
val Painter = participant3.parties.enable("Painter")

// Wait until the party enabling has taken effect and has been observed at the
↪participants
val partyAssignment = Set(Bank -> participant1, HouseOwner -> participant2,
↪Painter -> participant3)
participant2.parties.await_topology_observed(partyAssignment)
participant3.parties.await_topology_observed(partyAssignment)

// upload the Daml model to all participants
val darPath = Option(System.getProperty("canton-examples.dar-path")).getOrElse(
↪"dars/CantonExamples.dar")
participants.all.dars.upload(darPath)
```

Creating the IOU and the paint offer

To initialize the ledger, the Bank creates an IOU for the house owner and the house owner creates a paint offer for the painter. These steps are implemented below using the Scala bindings (no longer supported by Daml) generated from the Daml model. The generated Scala classes are distributed with the Canton release in the package `com.digitalasset.canton.examples`. The relevant classes are imported as follows:

```
import com.digitalasset.canton.examples.Iou.{Amount, Iou}
import com.digitalasset.canton.examples.Paint.{OfferToPaintHouseByOwner,
↪PaintHouse}
```

(continues on next page)

(continued from previous page)

```
import com.digitalasset.canton.participant.ledger.api.client.DecodeUtil.
↪decodeAllCreated
import com.digitalasset.canton.protocol.ContractIdSyntax._
```

Bank creates an IOU of USD 100 for the house owner on the `iou` domain, by *submitting the command* through the ledger API command service of participant 1. The house owner then shares the IOU contract with the painter such that the painter can effect the ownership change when they accept the offer. The share operation adds the painter as an observer on the IOU contract so that the painter can see the IOU contract. Both of these commands run over the `iou` domain because the Bank's participant 1 is only connected to the `iou` domain.

```
// Bank creates IOU for the house owner
val createIouCmd = Iou(
  payer = Bank.toPrim,
  owner = HouseOwner.toPrim,
  amount = Amount(value = 100.0, currency = "USD"),
  viewers = List.empty
).create.command
val Seq(iouContractUnshared) = decodeAllCreated(Iou)(
  participant1.ledger_api.commands.submit_flat(Seq(Bank), Seq(createIouCmd)))

// Wait until the house owner sees the IOU in the active contract store
participant2.ledger_api.acs.await_active_contract(HouseOwner, iouContractUnshared.
↪contractId.toLf)

// The house owner adds the Painter as an observer on the IOU
val shareIouCmd = iouContractUnshared.contractId.exerciseShare(actor = HouseOwner.
↪toPrim, viewer = Painter.toPrim).command
val Seq(iouContract) = decodeAllCreated(Iou)(participant2.ledger_api.commands.
↪submit_flat(Seq(HouseOwner), Seq(shareIouCmd)))
```

Similarly, the house owner creates a paint offer on the `paint` domain via participant 2. In the `ledger_api.commands.submit_flat` command, we set the workflow id to the `paint` domain so that the participant submits the commands to this domain. If no domain was specified, the participant automatically determines a suitable domain. In this case, both domains are eligible because on each domain, every stakeholder (the house owner and the painter) is hosted on a connected participant.

```
// The house owner creates a paint offer using participant 2 and the Paint domain
val paintOfferCmd = OfferToPaintHouseByOwner(
  painter = Painter.toPrim,
  houseOwner = HouseOwner.toPrim,
  bank = Bank.toPrim,
  iouId = iouContract.contractId
).create.command
val Seq(paintOffer) = decodeAllCreated(OfferToPaintHouseByOwner)(
  participant2.ledger_api.commands.submit_flat(Seq(HouseOwner),
↪Seq(paintOfferCmd), workflowId = paint.name))
```

Contracts and Their Domains

In Canton, each contract is only known to the participants involved in that contract. The involved participants are the only ones that have unencrypteded copies of the contract, which they store in their respective private contract stores. No other participant has access to that data, even in encrypted form. The domain, in particular the sequencer that facilitates synchronization, will only store encrypted messages that only the receiving participant can decrypt.

In our terminology, the residence domain of a contract is the current agreement between the stakeholders of the contract where changes to the contract are to be communicated and where the sequence of actions on a contract is to be determined. A contract can reside on at most one domain at any point in time. However, the contract is never stored by the domain in such a way that the domain learns about its existence or content.

Transferring a contract

For example, the IOU contract resides on the `iou` domain because it has been created by a command that was submitted to the `iou` domain. Similarly, the paint offer resides on the `paint` domain. In the current version of Canton, the execution of a transaction can only use contracts that reside on a single domain. Therefore, before the painter can accept the offer and thereby become the owner of the IOU contract, both contracts must be brought to a common domain.

In this example, the house owner and the painter are hosted on participants that are connected to both domains, whereas the Bank is only connected to the `iou` domain. The IOU contract cannot be moved to the `paint` domain because all stakeholders of a contract must be connected to the contract's domain of residence. Conversely, the paint offer can be transferred to the `iou` domain, so that the painter can accept the offer on the `iou` domain.

Stakeholders can change the residence domain of a contract using the `transfer.execute` command. In the example, the painter transfers the paint offer from the `paint` domain to the `iou` domain.

```
// Wait until the painter sees the paint offer in the active contract store
participant3.ledger_api.acs.await_active_contract(Painter, paintOffer.contractId.
↪toLf)

// Painter transfers the paint offer to the IOU domain
participant3.transfer.execute(
  Painter,                      // Initiator of the transfer
  paintOffer.contractId.toLf,   // Contract to be transferred
  paintAlias,                   // Source domain
  iouAlias                      // Target domain
)
```

The transfer of a contract effectively changes the residence domain of the contract, in other words, the consensus among the stakeholders on which domain should be used to sequence actions on a contract. The contract itself is still stored only on the involved participants.

Atomic acceptance

The paint offer and the IOU contract both reside on the `iou` domain now. Accordingly, the painter can complete the workflow by accepting the offer.

```
// Painter accepts the paint offer on the IOU domain
val acceptCmd = paintOffer.contractId.exerciseAcceptByPainter(Painter.toPrim).
↪command
val acceptTx = participant3.ledger_api.commands.submit_flat(Seq(Painter),
↪Seq(acceptCmd))
val Seq(painterIou) = decodeAllCreated(Iou)(acceptTx)
val Seq(paintHouse) = decodeAllCreated(PaintHouse)(acceptTx)
```

This transaction executes on the `iou` domain because the input contracts (the paint offer and the IOU) reside there. It atomically creates two contracts on the `iou` domain: the painter's new IOU and the agreement to paint the house. The unhappy scenarios needing out-of-band resolution are avoided.

Completing the workflow

Finally, the paint agreement can be transferred back to the `paint` domain, where it actually belongs.

```
// Wait until the house owner sees the PaintHouse agreement
participant2.ledger_api.acs.await_active_contract(HouseOwner, paintHouse.
↪contractId.toLf)

// The house owner moves the PaintHouse agreement back to the Paint domain
participant2.transfer.execute(
  HouseOwner,
  paintHouse.contractId.toLf,
  iouAlias,
  paintAlias
)
```

Note that the painter's IOU remains on the `iou` domain. The painter can therefore call the IOU and cash it out.

```
// Painter converts the Iou into cash
participant3.ledger_api.commands.submit_flat(
  Seq(Painter),
  Seq(painterIou.contractId.exerciseCall(Painter.toPrim).command),
  iou.name
)
```

Performing transfers automatically

Canton also supports automatic transfers for commands performing transactions that use contracts residing on several domains. When such a command is submitted, Canton can automatically infer a common domain that the used contracts can be transferred to. Once all the used contracts have been transferred into the common domain the transaction is performed on this single domain. However, this simply performs the required transfers followed by the transaction processing as distinct non-atomic steps.

We can therefore run the above script without specifying any transfers at all, and relying on the automatic transfers. Simply delete all the transfer commands from the example above and the example will still run successfully. A modified version of the above example that uses automatic transfers instead of manual transfers is given below.

The setup code and contract creation is unchanged:

```
// Bank creates IOU for the house owner
val createIouCmd = Iou(
  payer = Bank.toPrim,
  owner = HouseOwner.toPrim,
  amount = Amount(value = 100.0, currency = "USD"),
  viewers = List.empty
).create.command
val Seq(iouContractUnshared) = decodeAllCreated(Iou)(
  participant1.ledger_api.commands.submit_flat(Seq(Bank), Seq(createIouCmd)))

// Wait until the house owner sees the IOU in the active contract store
participant2.ledger_api.acs.await_active_contract(HouseOwner, iouContractUnshared.
↪contractId.toLf)

// The house owner adds the Painter as an observer on the IOU
val showIouCmd = iouContractUnshared.contractId.exerciseShare(actor = HouseOwner.
↪toPrim, viewer = Painter.toPrim).command
val Seq(iouContract) = decodeAllCreated(Iou)(participant2.ledger_api.commands.
↪submit_flat(Seq(HouseOwner), Seq(showIouCmd)))

// The house owner creates a paint offer using participant 2 and the Paint domain
val paintOfferCmd = OfferToPaintHouseByOwner(
  painter = Painter.toPrim,
  houseOwner = HouseOwner.toPrim,
  bank = Bank.toPrim,
  iouId = iouContract.contractId
).create.command
val Seq(paintOffer) = decodeAllCreated(OfferToPaintHouseByOwner)(
  participant2.ledger_api.commands.submit_flat(Seq(HouseOwner),
↪Seq(paintOfferCmd), workflowId = paint.name))
```

In the following section, the painter accepts the paint offer. The transaction that accepts the paint offer uses two contracts: the paint offer contract, and the IOU contract. These contracts were created on two different domains in the previous step: the paint offer contract was created on the paint domain, and the IOU contract was created on the IOU domain. The paint offer contract must be transferred to the IOU domain for the accepting transaction to be successfully applied, as was done manually in the example above. It would not be possible to instead transfer the IOU contract to the paint domain because the stakeholder Bank on the IOU contract is not represented on the paint domain.

When using automatic-transfer transactions, Canton infers a suitable domain for the transaction

and transfers all used contracts to this domain before applying the transaction. In this case, the only suitable domain for the painter to accept the paint offer is the IOU domain. This is how the painter is able to accept the paint offer below without any explicit transfers being performed.

```
// Wait until the painter sees the paint offer in the active contract store
participant3.ledger_api.acs.await_active_contract(Painter, paintOffer.contractId.
↪toLf)

// Painter accepts the paint offer on the IOU domain
val acceptCmd = paintOffer.contractId.exerciseAcceptByPainter(Painter.toPrim).
↪command
val acceptTx = participant3.ledger_api.commands.submit_flat(Seq(Painter),
↪Seq(acceptCmd))
val Seq(painterIou) = decodeAllCreated(Iou)(acceptTx)
val Seq(paintHouse) = decodeAllCreated(PaintHouse)(acceptTx)
```

The painter can then cash in the IOU. This happens exactly as before, since the IOU contract never leaves the IOU domain.

```
// Painter converts the Iou into cash
participant3.ledger_api.commands.submit_flat(
  Seq(Painter),
  Seq(painterIou.contractId.exerciseCall(Painter.toPrim).command),
  iou.name
)
```

Note that towards the end of the previous example with explicit transfers, the paint offer contract was transferred back to the paint domain. This doesn't happen in the automatic transfer version: the paint offer is not transferred out of the IOU domain as part of the script shown. However, the paint offer contract will be automatically transferred back to the paint domain once it is used in a transaction that must happen on the paint domain.

Details of the automatic-transfer transactions

In the previous section, the automatic-transfer transactions were explained using an example. The details are presented here.

The automatic-transfer transactions enable submission of a transaction using contracts on multiple domains, by transferring contracts into a chosen target domain and then performing the transaction. However, using an automatic-transfer transaction does not provide any atomicity guarantees beyond using several primitive transfer-in and transfer-out operations (these operations make up the `transfer.execute` command, and are explained in the next section).

The domain for a transaction is chosen using the following criteria:

> Minimise the number of transfers needed.
> Break ties by choosing domains with higher priority first.
> Break ties by choosing domains with alphabetically smaller domain IDs first.

As for ordinary transactions, you may force the choice of domain for an automatic-transfer transaction by setting the workflow ID to name of the domain.

The automatic-transfer transactions are only enabled when all of the following are true:

> The local canton console enables preview commands (see the *configuration* section).

The submitting participant is connected to all domains that contracts used by the transaction live on.
All contracts used by the transaction must have at least one stakeholder that is also a transaction submitter.

Take aways

A contract resides on a domain. This means that the current agreement of the stakeholders is to communicate and sequence all access and changes to a given contract on a particular domain. The contract itself is only stored at the stakeholder participants.
Stakeholders can move contracts from one domain to another using `transfer.execute`. All stakeholders must be connected to the source and the target domain.
You can submit transactions using contracts that reside on several domains. Automatic transfers will pick a suitable domain, and perform the transfers into it before performing the transaction.

1.23.6.2 Part 2: Composing existing workflows

This part shows how existing workflows can be composed even if they work on separate domains. The running example is a variation of the paint example from the first part with a more complicated topology. We therefore assume that you have gone through *the first part* of this tutorial. Technically, this tutorial runs through the same steps as the first part, but more details are exposed. The console commands assume that you start with a fresh Canton console.

Existing workflows

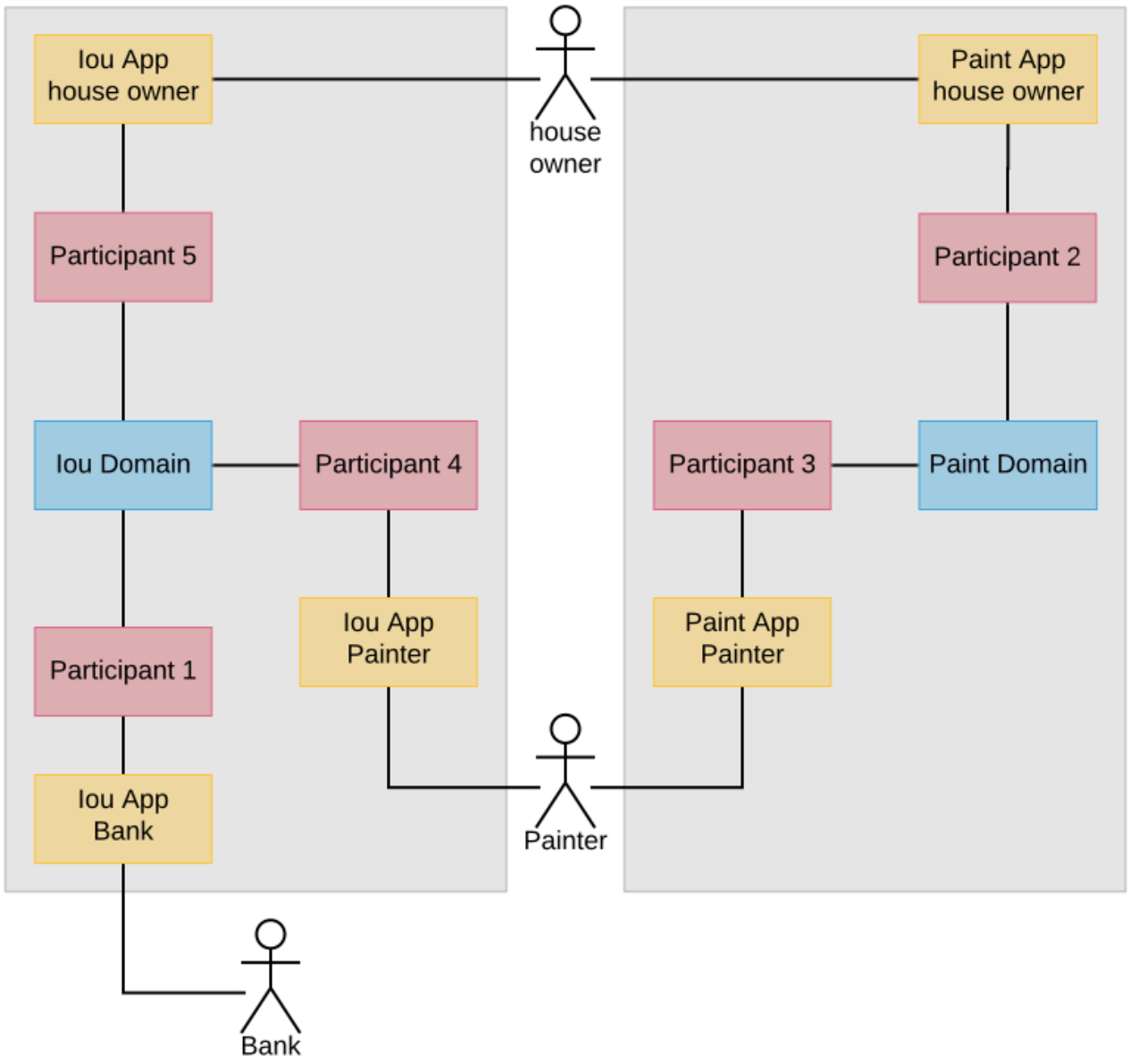
Consider a situation where the two domains `iou` and `paint` have evolved separately:

The `iou` domain for managing IOUs,
The `paint` domain for managing paint agreements.

Accordingly, there are separate applications for managing IOUs (issuing, changing ownership, calling) and paint agreements, and the house owner and the painter have connected their applications to different participants. The situation is illustrated in the following picture.

To enter in a paint agreement in this setting, the house owner and the painter need to perform the following steps:

1. The house owner creates a paint offer through participant 2 on the `paint` domain.
2. The painter accepts the paint offer through participant 3 on the `paint` domain. As a consequence, a paint agreement is created.
3. The painter sets a reminder that he needs to receive an IOU from the house owner on the `iou` domain.
4. When the house owner observes a new paint agreement through participant 2 on the `paint` domain, she changes the IOU ownership to the painter through participant 5 on the `iou` domain.
5. The painter observes a new IOU through participant 4 on the `iou` domain and therefore removes the reminder.

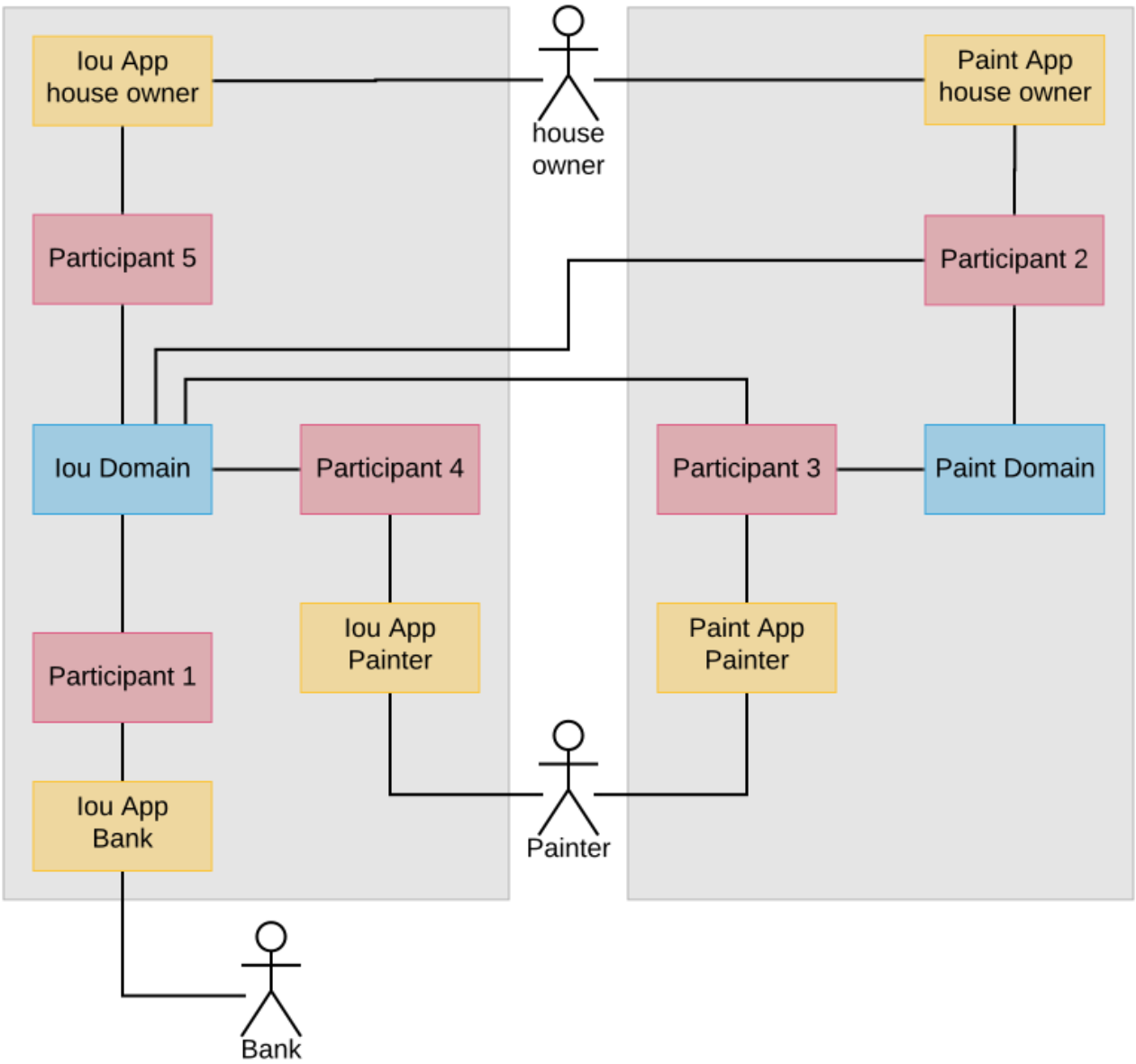

Iou App
house owner
house
owner
Paint App
house owner
Participant 5
Participant 2
Iou Domain
Participant 4
Participant 3
Paint Domain
Iou App
Painter
Paint App
Painter
Participant 1
Iou App
Bank
Painter
Bank

Overall, a non-trivial amount of out-of-band coordination is required to keep the `paint` ledger consistent with the `iou` ledger. If this coordination breaks down, the *unhappy scenarios from the first part* can happen.

Required changes

We now show how the house owner and the painter can avoid need for out-of-band coordination when entering in paint agreements. The goal is to reuse the existing infrastructure for managing IOUs and paint agreements as much as possible. The following changes are needed:

1. The house owner and the painter connect their participants for paint agreements to the `iou` domain:



   The *Canton configuration* is accordingly extended with the two participants 4 and 5. (The connections themselves are set up in the *next section*.)

```
canton {
  participants {
    participant4 {
      ledger-api.port = 13041
      admin-api.port = 13042
      storage.type = memory
      init.parameters.unique-contract-keys = false
    }

    participant5 {
      ledger-api.port = 13051
      admin-api.port = 13052
      storage.type = memory
      init.parameters.unique-contract-keys = false
    }
  }
}
```

2. They replace their Daml model for paint offers such that the house owner must specify an IOU in the offer and its accept choice makes the painter the new owner of the IOU.
3. They create a new application for the *paint offer-accept workflow*.

The Daml models for IOUs and paint agreements themselves remain unchanged, and so do the applications that deal with them.

Preparation using the existing workflows

We extend the topology from the first part as described. The commands are explained in detail in Canton's *identity management manual*.

```
// update parameters
iou.service.update_dynamic_domain_parameters(
  _.update(transferExclusivityTimeout = Duration.Zero)
) // disable automatic transfer-in

paint.service.update_dynamic_domain_parameters(
  _.update(transferExclusivityTimeout = 2.seconds)
)

// connect participants to the domain
participant1.domains.connect_local(iou)
participant2.domains.connect_local(iou)
participant3.domains.connect_local(iou)
participant2.domains.connect_local(paint)
participant3.domains.connect_local(paint)
participant4.domains.connect_local(iou)
participant5.domains.connect_local(iou)

val iouAlias = iou.name
val paintAlias = paint.name

// create the parties
val Bank = participant1.parties.enable("Bank")
val HouseOwner = participant2.parties.enable("House Owner")
val Painter = participant3.parties.enable("Painter", waitForDomain = DomainChoice.
↪All)
```

(continues on next page)

(continued from previous page)

```
// enable the house owner on participant 5 and the painter on participant 4
// as explained in the identity management documentation at
// https://docs.daml.com/canton/usermanual/identity_management.html#party-on-two-
↪nodes
import com.digitalasset.canton.console.ParticipantReference
def authorizePartyParticipant(partyId: PartyId, createdAt: ParticipantReference,
↪to: ParticipantReference): Unit = {
  val createdAtP = createdAt.id
  val toP = to.id
  createdAt.topology.party_to_participant_mappings.authorize(TopologyChangeOp.Add,
↪ partyId, toP, RequestSide.From)
  to.topology.party_to_participant_mappings.authorize(TopologyChangeOp.Add,
↪partyId, toP, RequestSide.To)
}
authorizePartyParticipant(HouseOwner, participant2, participant5)
authorizePartyParticipant(Painter, participant3, participant4)

// Wait until the party enabling has taken effect and has been observed at the
↪participants
val partyAssignment = Set(HouseOwner -> participant2, HouseOwner -> participant5,
↪Painter -> participant3, Painter -> participant4)
participant2.parties.await_topology_observed(partyAssignment)
participant3.parties.await_topology_observed(partyAssignment)

// upload the Daml model to all participants
val darPath = Option(System.getProperty("canton-examples.dar-path")).getOrElse(
↪"dars/CantonExamples.dar")
participants.all.dars.upload(darPath)
```

As before, the Bank creates an IOU and the house owner shares it with the painter on the `iou` domain, using their existing applications for IOUs.

```
import com.digitalasset.canton.examples.Iou.{Amount, Iou}
import com.digitalasset.canton.examples.Paint.{OfferToPaintHouseByOwner,
↪PaintHouse}
import com.digitalasset.canton.participant.ledger.api.client.DecodeUtil.
↪decodeAllCreated
import com.digitalasset.canton.protocol.ContractIdSyntax._

val createIouCmd = Iou(
  payer = Bank.toPrim,
  owner = HouseOwner.toPrim,
  amount = Amount(value = 100.0, currency = "USD"),
  viewers = List.empty
).create.command
val Seq(iouContractUnshared) = decodeAllCreated(Iou)(
  participant1.ledger_api.commands.submit_flat(Seq(Bank), Seq(createIouCmd)))

// Wait until the house owner sees the IOU in the active contract store
participant2.ledger_api.acs.await_active_contract(HouseOwner, iouContractUnshared.
↪contractId.toLf)

// The house owner adds the Painter as an observer on the IOU
val shareIouCmd = iouContractUnshared.contractId.exerciseShare(actor = HouseOwner.
↪toPrim, viewer = Painter.toPrim).command
```

(continues on next page)

(continued from previous page)

```
val Seq(iouContract) = decodeAllCreated(Iou)(participant2.ledger_api.commands.
↪submit_flat(Seq(HouseOwner), Seq(shareIouCmd)))
```

The paint offer-accept workflow

The new paint offer-accept workflow happens in four steps:

1. Create the offer on the `paint` domain.
2. Transfer the contract to the `iou` domain.
3. Accept the offer.
4. Transfer the paint agreement to the `paint` domain.

Making the offer

The house owner creates a paint offer on the `paint` domain.

```
// The house owner creates a paint offer using participant 2 and the Paint domain
val paintOfferCmd = OfferToPaintHouseByOwner(
  painter = Painter.toPrim,
  houseOwner = HouseOwner.toPrim,
  bank = Bank.toPrim,
  iouId = iouContract.contractId
).create.command
val Seq(paintOffer) = decodeAllCreated(OfferToPaintHouseByOwner)(
  participant2.ledger_api.commands.submit_flat(Seq(HouseOwner),
↪Seq(paintOfferCmd), workflowId = paint.name))
```

Transfers are not atomic

In the first part, we have used `transfer.execute` to move the offer to the `iou` domain. Now, we look a bit behind the scenes. A contract transfer happens in two atomic steps: transfer-out and transfer-in. `transfer.execute` is merely a shorthand for the two steps. In particular, `transfer.execute` is not an atomic operation like other ledger commands.

During a transfer-out, the contract is deactivated on the source domain, in this case the `paint` domain. Any stakeholder whose participant is connected to the source domain and the target domain can initiate a transfer-out. The `transfer.out` command returns a transfer Id.

```
// Wait until the painter sees the paint offer in the active contract store
participant3.ledger_api.acs.await_active_contract(Painter, paintOffer.contractId.
↪toLf)

// Painter transfers the paint offer to the IOU domain
val paintOfferTransferId = participant3.transfer.out(
  Painter,                      // Initiator of the transfer
  paintOffer.contractId.toLf,   // Contract to be transferred
  paintAlias,                   // Source domain
  iouAlias                      // Target domain
)
```

The `transfer.in` command consumes the transfer Id and activates the contract on the target domain.

```
participant3.transfer.in(Painter, paintOfferTransferId, iouAlias)
```

Between the transfer-out and the transfer-in, the contract does not reside on any domain and cannot be used by commands. We say that the contract is in transit.

Accepting the paint offer

The painter accepts the offer, as before.

```
// Wait until the Painter sees the IOU contract on participant 3.
participant3.ledger_api.acs.await_active_contract(Painter, iouContract.contractId.
↪toLf)

// Painter accepts the paint offer on the Iou domain
val acceptCmd = paintOffer.contractId.exerciseAcceptByPainter(Painter.toPrim).
↪command
val acceptTx = participant3.ledger_api.commands.submit_flat(Seq(Painter),
↪Seq(acceptCmd))
val Seq(painterIou) = decodeAllCreated(Iou)(acceptTx)
val Seq(paintHouse) = decodeAllCreated(PaintHouse)(acceptTx)
```

Automatic transfer-in

Finally, the paint agreement is transferred back to the `paint` domain such that the existing infrastructure around paint agreements can work unchanged.

```
// Wait until the house owner sees the PaintHouse agreement
participant2.ledger_api.acs.await_active_contract(HouseOwner, paintHouse.
↪contractId.toLf)

val paintHouseId = paintHouse.contractId
// The house owner moves the PaintHouse agreement back to the Paint domain
participant2.transfer.out(
  HouseOwner,
  paintHouseId.toLf,
  iouAlias,
  paintAlias
)
// After the exclusivity period, which is set to 2 seconds,
// the contract is automatically transferred into the target domain
utils.retry_until_true(10.seconds) {
    // in the absence of other activity, force the participants to update their
↪view of the latest domain time
    participant2.testing.fetch_domain_times()
    participant3.testing.fetch_domain_times()

    participant3.testing.acs_search(paint.name, filterId=paintHouseId.toString).
↪nonEmpty &&
      participant2.testing.acs_search(paint.name, filterId=paintHouseId.toString).
↪nonEmpty
}
```

Here, there is only a `transfer.out` command but no `transfer.in` command. This is because the participants of contract stakeholders automatically try to transfer-in the contract to the target domain so that the contract becomes usable again. The domain parameter `transfer-exclusivity-timeout` on the target domain specifies how long they wait before they attempt to do so. Before the timeout, only the initiator of the transfer is allowed to transfer-in the contract. This reduces contention for contracts with many stakeholders, as the initiator normally completes the transfer before all other stakeholders simultaneously attempt to transfer-in the contract. On the `paint` domain, this timeout is set to two seconds in the *configuration* file. Therefore, the `utils.retry_until_true` normally succeeds within the allotted ten seconds.

Setting the `transfer-exclusivity-timeout` to 0 as on the `iou` domain disables automatic transfer-in. This is why the above transfer of the paint offer had to be completed manually. Manual completion is also needed if the automatic transfer-in fails, e.g., due to timeouts on the target domain. Automatic transfer-in therefore is a safety net that reduces the risk that the contract gets stuck in transit.

Continuing the existing workflows

The painter now owns an IOU on the `iou` domain and the entered paint agreement resides on the `paint` domain. Accordingly, the existing workflows for IOUs and paint agreements can be used unchanged. For example, the painter can call the IOU.

```
// Painter converts the Iou into cash
participant4.ledger_api.commands.submit_flat(
  Seq(Painter),
  Seq(painterIou.contractId.exerciseCall(Painter.toPrim).command),
  iou.name
)
```

Take aways

Contract transfers take two atomic steps: transfer-out and transfer-in. While the contract is being transferred, the contract does not reside on any domain.
Transfer-in happens under normal circumstances automatically after the `transfer-exclusivity-timeout` configured on the target domain. A timeout of 0 disables automatic transfer-in. If the automatic transfer-in does not complete, the contract can be transferred in manually.

1.23.7 Versioning

1.23.7.1 Canton release version

The Canton release version (release version for short) is the primary version assigned to a Canton release. It is semantically versioned, i.e., breaking changes to a public API will always lead to a major version increase of the release version. The public APIs encompassed by the release version are the following:

Ledger API server (for participants)
Error code format (machine-readable parts, see also *the error code documentation*)
Canton configuration file format

Command line arguments
Internal storage (data continuity between non-major upgrades)
Canton protocol version

As a result, Canton components are always safely upgradeable with respect to these APIs. In particular, the inclusion of the Canton protocol version as a Public API guarantees that any two Canton components of the same release version can interact with each other and can be independently upgraded within a major version without any loss of interoperability (see also *the documentation on the Canton protocol version*).

1.23.7.2 For application developers and operators

Applications using Canton have the following guarantees:

Participants can be upgraded independently of each other and of applications and domains within a major release version.
Domain drivers can be upgraded independently of applications and connected participants within a major release version.
Major versions of anything are supported for a minimum of 12 months from the release of the next major release version.

As a result, applications written today can keep running unchanged for a minimum of 12 months while upgrading participants and domains within a major release version. See also the versioning as well as portability, compatibility and support duration guarantees that hold for any Daml application.

1.23.7.3 For Canton participant and domain operators

In addition to the Canton release version, the Canton protocol version is the most important version for participant and domain operators. It used to have 3 digits, but starting protocol version *4* it's represented by one digit.

Canton protocol version

The Canton protocol determines how different Canton components interact with each other. We version it using the Canton protocol version (protocol version for short) and conceptually, two Canton components can interact (are interoperable) if they support the same protocol version. For example, a participant can connect to a domain if it supports the protocol version that is spoken on the domain, and a mediator can become the mediator for a domain, if it supports the protocol version required by the domain. If two Canton components have the same major release version, they also share at least one protocol version and can thus interact with each other.

A Canton component advertises the protocol versions it supports and always supports all previous protocol versions of the same major release line. That is, a participant or driver supporting a certain protocol version, is able to transact with all other participants or drivers supporting a lower or equal protocol version but may not be able to transact with participants or drivers supporting a higher Canton protocol if they are configured to use a more recent version of the protocol. For example, a release of a participant supporting protocol version *3* will be able to connect to all domains configured to use protocol version <= *3*. It won't be able to connect to a domain configured to use protocol version > *3*. As a result, minor and patch version upgrades of Canton components can be done independently without any loss of interoperability.

To see the protocol versions a Canton component supports (e.g., *2* and *3*), run

```
canton --version
```

(where `canton` is an alias for the path pointing to the Canton release binary `bin/canton`).

The list of supported protocol versions for each minor version is the following:

Release	Protocol versions
2.0	2
2.1	2
2.2	2
2.3	2, 3
2.4	2, 3
2.5	2, 3, 4
2.6	3, 4
2.7	3, 4, 5

Features and protocol versions

Some Canton features are only available on domains running specific protocol versions. The following table indicates the protocol versions required to use some features.

Protocol version	Feature
4 and above	Interfaces

Configuring the protocol version

A Canton driver or domain operator is able to configure the protocol version spoken on the domain (e.g. *3*). If the domain operator sets the protocol version spoken on a domain too high, they may exclude participants that don't support this protocol version yet.

For example, if the domain operator sets the protocol version on a domain to 3, participants that only support protocol version 2 aren't able to connect to the domain. They would be able to connect and transact on the domain, if the protocol version set on the domain is set to 2 or lower. Note that if the participant and domain come from the same major release line, the domain will also support using protocol version 2. Thus, the domain could be configured such that the participant could connect to it

Minimum protocol version

Similar to how a domain operator is able to configure the protocol version spoken on a domain, a participant operator is able to configure a minimum protocol version for a participant. Configuring a minimum protocol version guarantees that a participant will only connect to domain that use at least this protocol version or a newer one. This is especially desirable to ensure that a participant only connects to domains that have certain security patches applied or that support particular protocol features.

Support and bug fixes

Canton protocol major versions are supported for a minimum of 12 months from the release of the next major version. Within a major version, only the latest minor version receives security and bug fixes.

1.24 Obtaining Canton

1.24.1 Choosing Open-Source or Enterprise Edition

The Canton distributed ledger is included with Daml Enterprise edition, or available as open source. Which role the application takes depends on the configuration. The main administration interface of the Canton application is the embedded console, which is part of the application.

Canton releases come in two variants: Open-Source or Enterprise. Both support the full Canton protocol, but differ in terms of enterprise and non-functional capabilities:

Table 2: Differences between Enterprise and Open Source Edition

Capability	Enterprise	Open-Source
Daml Synchronisation	Yes	Yes
Sub-Transaction Privacy	Yes	Yes
Transaction Processing	Parallel (fast)	Sequential (slow)
High Availability	*Yes*	No
High Throughput via Microservices	*Yes*	No
Resource Management	Yes	No
Ledger Pruning	Yes	No
AWS KMS support	Yes	No
Postgres Backend	Yes	Yes
Oracle Backend	Yes	No
Besu driver	Yes	No
Fabric driver	Yes	No

Please follow below instructions in order to obtain your copy of Canton.

1.24.2 Downloading the Open Source Edition

The Open Source release is available from Github. You can also use our Canton Docker images by following our *Docker instructions*.

1.24.3 Downloading the Enterprise Edition

Enterprise releases are available on request (sales@digitalasset.com) and can be downloaded from the respective repository, or you can use our Canton Enterprise Docker images as described in our *Docker instructions*.

1.24.4 Installing Canton

This guide will guide you through the process of setting up your Canton nodes to build a distributed Daml ledger. You will learn

1. How to setup and configure a domain
2. How to setup and configure one or more participant nodes

Note: As no topology is the same, this guide will point out different configuration options as notes wherever possible.

This guide uses the example configurations you can find in the release bundle under `example/03-advanced-configuration` and explains you how to leverage these examples for your purposes. Therefore, any file named in this guide will refer to subdirectories of the advanced configuration example.

If you are using Oracle JVM and testing security provider signatures, you should note that the provided Canton JAR file embeds the BouncyCastle Provider as a dependency. To enable the JVM to verify the signature, you need to put the `bcprov` JAR on the classpath before the Canton Standalone JAR. For example:

```
java -cp bcprov-jdk15on-1.70.jar:canton-with-drivers-2.7.4-all.jar com.
↪digitalasset.canton.CantonEnterpriseApp
```

1.24.4.1 Downloading Canton

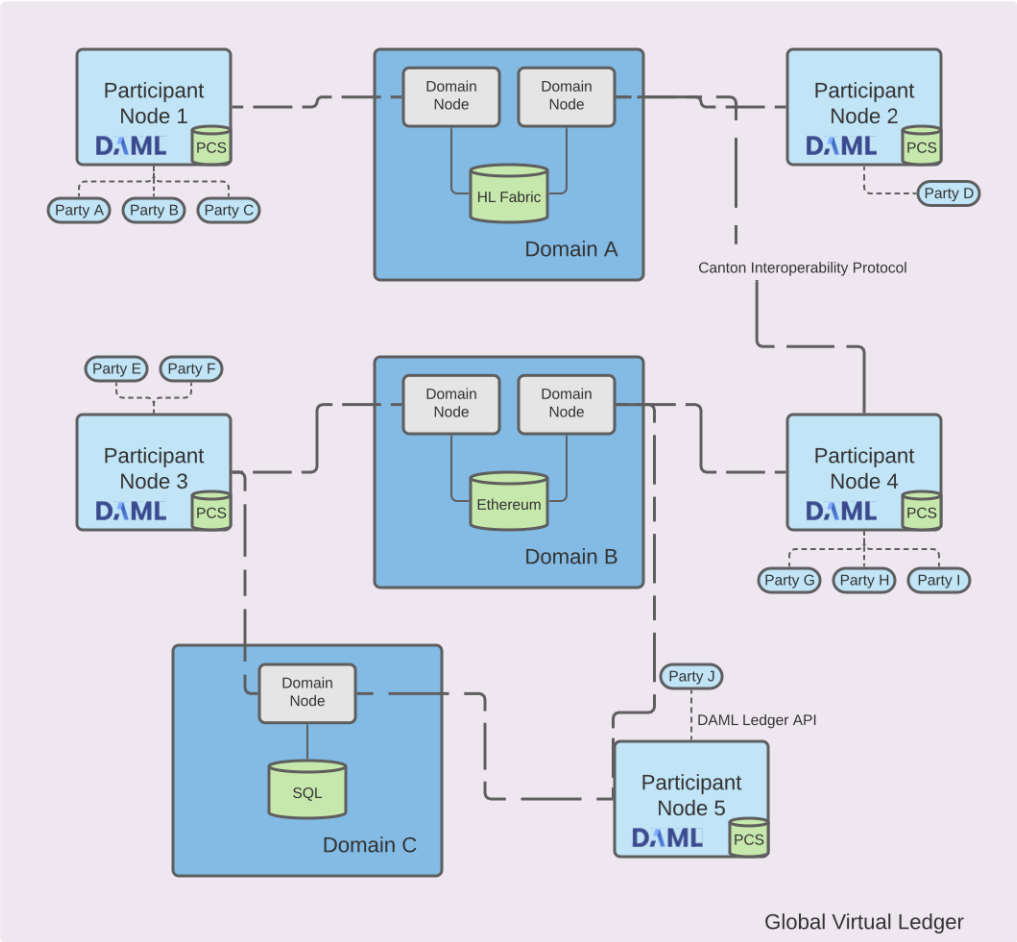
The Canton Open Source code is available from Github. You can also use our Canton Docker images by following our *Docker instructions*.

Daml Enterprise includes an enterprise version of the Canton ledger. If you have entitlement to Daml Enterprise you can download the enterprise version of Canton by following the Installing Daml Enterprise instructions and downloading the appropriate Canton artifact.

1.24.4.2 Your Topology

The first question we need to address is what the topology is that you are going after. The Canton topology is made up of parties, participants and domains, as depicted in the following figure.

The Daml code will run on the participant node and expresses smart contracts between parties. Parties are hosted on participant nodes. Participant nodes will synchronise their state with other participant nodes by exchanging messages with each other through domains. Domains are nodes that integrate with the underlying storage technology such as databases or other distributed ledgers. As the Canton protocol is written in a way that assumes that Participant nodes don't trust each other,


Participant Node 1
DAML
PCS
Party A
Party B
Party C
Domain Node
Domain Node
HL Fabric
Domain A
Participant Node 2
DAML
PCS
Party D
Canton Interoperability Protocol
Party E
Party F
Participant Node 3
DAML
PCS
Domain Node
Domain Node
Ethereum
Domain B
Participant Node 4
DAML
PCS
Party G
Party H
Party I
Domain Node
SQL
Domain C
Party J
DAML Ledger API
Participant Node 5
DAML
PCS
Global Virtual Ledger

you would normally expect that every organisation runs only one participant node, except for scaling purposes.

If you want to build up a test-network for yourself, you need at least a participant node and a domain.

1.24.4.3 Environment Variables

For our convenience in this guide, we will use a few environment variables to refer to a set of directions. Please set the environment variable CANTON to point to the place where you have unpacked the canton release bundle.

```
cd ./canton-X.Y.Z
export CANTON=`pwd`
```

And then set another variable that points to the advanced example directory

```
export CONF="$CANTON/examples/03-advanced-configuration"
```

1.24.4.4 Selecting your Storage Layer

In order to run any kind of node, you need to decide how and if you want to persist the data. You currently have three choices: don't persist and just use in-memory stores which will be deleted if you restart your node or persist using `Postgres` or `Oracle` databases.

For this purpose, there are some storage *mixin configurations* (`storage/`) defined. These storage mixins can be used with any of the node configurations. The in-memory configurations just work out of the box without further configuration. The database based persistence will be explained in a subsequent section, as you first need to initialise the database.

The mixins work by defining a shared variable which can be referenced by any node configuration

```
storage = ${_shared.storage}
storage.parameters.databaseName = "participant1"
```

If you ever see the following error: `Could not resolve substitution to a value: ${_shared.storage}`, then you forgot to add the persistence mixin configuration file.

Note: Please also consult the more *detailed section on persistence configurations*.

Persistence using Postgres

While in-memory is great for testing and demos, for more serious tasks, you need to use a database as a persistence layer. Both the community version and the enterprise version support Postgres as a persistence layer. Make sure that you have a running Postgres server and you need to create one database per node. The recommended Postgres version to use is 11, as this is tested the most thoroughly.

The Postgres storage mixin is provided by the file `storage/postgres.conf`.

If you just want to experiment, you can use Docker to get a Postgres database up and running quickly. Here are a few commands that come in handy.

First, pull Postgres and start it up.

```
docker pull postgres:14.8-bullseye
docker run --rm --name pg-docker -e POSTGRES_PASSWORD=docker -d -p 5432:5432
↪postgres:14.8-bullseye
```

Then, you can run `psql` using:

```
docker exec -it pg-docker psql -U postgres -d postgres
```

This will invoke `psql` interactively. You can exit the prompt with Ctrl-D. If you want to just cat commands, change `-it` to `-i` in above command.

Then, create a user for the database using the following SQL command

```
create user canton with encrypted password 'supersafe';
```

and create a new database for each node, granting the newly created user appropriate permissions

```
create database participant1;
grant all privileges on database participant1 to canton;
```

These commands create a database named `participant1` and grant the user named `canton` access to it using the password `supersafe`. Needless to say, you should use your own, secure password.

In order to use the storage mixin, you need to either write these settings into the configuration file, or pass them using environment variables:

```
export POSTGRES_USER=canton
export POSTGRES_PASSWORD=supersafe
```

If you want to run also other nodes with Postgres, you need to create additional databases, one for each.

You can reset the database by dropping then re-creating it:

```
drop database participant1;
create database participant1;
grant all privileges on database participant1 to canton;
```

Note: The storage mixin provides you with an initial configuration. Please consult the more *extended documentation* for further options.

If you are setting up a few nodes for a test network, you can use a little helper script to create the SQL commands to setup users and databases:

```
python3 examples/03-advanced-configuration/storage/dbinit.py \
   --type=postgres --user=canton --password=<choose-wisely> --participants=2 --
↪domains=1 --drop
```

The command will just create the SQL commands for your convenience. You can pipe the output directly into the `psql` command

```
python3 examples/03-advanced-configuration/storage/dbinit.py ... | psql -p 5432 -
↪h localhost ...
```

1.24.4.5 Setting up a Participant

Now that you have made your persistence choice (assuming Postgres hereafter, for Oracle refer to *Oracle Persistence*), you could start your participant just by using one of the example files such as `$CONF/nodes/participant1.conf` and start the Canton process using the `Postgres` persistence mixin:

```
$CANTON/bin/canton -c $CONF/storage/postgres.conf -c $CONF/nodes/participant1.conf
```

While this would work, we recommend that you rename your node by changing the configuration file appropriately.

Note: By default, the node will initialise itself automatically using the identity commands *Topology Administration*. As a result, the node will create the necessary keys and topology transactions and will initialise itself using the name used in the configuration file. Please consult the *identity management section* for further information.

This was everything necessary to startup your participant node. However, there are a few steps that you want to take care of in order to secure the participant and make it usable.

Secure the APIs

1. By default, all APIs in Canton are only accessible from localhost. If you want to connect to your node from other machines, you need to bind to `0.0.0.0` instead of localhost. You can do this by setting `address = 0.0.0.0` within the respective API configuration sections or include the `api/public.conf` configuration mixin.
2. The participant node is managed through the administration API. If you use the console, almost all requests will go through the administration API. We recommend that you setup mutual TLS authentication as described in the *TLS documentation section*.
3. Applications and users will interact with the participant node using the ledger API. We recommend that you secure your API by using TLS. You should also authorize your clients using either JWT or TLS client certificates. The TLS configuration is the same as on the administration API.
4. In the example set, there are a set of additional configuration options which allow you to define various JWT based authorizations checks, enforced by the ledger API server. The settings map exactly to the options documented as part of the Daml SDK. There are a few configuration mix-ins defined in `api/jwt` for your convenience.

Configure Applications, Users and Connection

Canton distinguishes static from dynamic configuration.

> *Static configuration* are items which are not supposed to change and are therefore captured in the configuration file. An example is to which port to bind to.
> Dynamic configuration are items such as Daml archives (DARs), domain connections or parties. All such changes are effected through *console commands* (or the *administration APIs*).

If you don't know how to connect to domains, onboard parties or provision Daml code, please read the *getting started guide*.

1.24.4.6 Setting up a Domain

In order to setup a domain, you need to decide what kind of domain you want to run. We provide integrations for different domain infrastructures. These integrations have different levels of maturity. Your current options are

1. Postgres based domain (simplest choice)
2. *Oracle based domain*
3. Hyperledger Fabric based domain
4. Ethereum based domain (demo)

This section will explain how to setup an in-process based domain using Postgres. All other domains are a set of microservices and part of the Enterprise edition. In any case, you will need to operate the main domain process which is the point of contact where participants connect to for the initial handshake and parameter download. The details of how to set this up for other domains than the in-process based Postgres domain are covered by the individual documentations.

Note: Please contact us at sales@digitalasset.com to get access to the Fabric or Ethereum based integration.

The domain requires independent of the underlying ledger a place to store some governance data (or also the messages in transit in the case of Postgres based domains). The configuration settings for this storage are equivalent to the settings used for the participant node.

Once you have picked the storage type, you can start the domain using

```
$CANTON/bin/canton -c $CONF/storage/postgres.conf -c $CONF/nodes/domain1.conf
```

Secure the APIs

1. As with the participant node, all APIs bind by default to localhost. You need to bind to `0.0.0.0` if you want to access the APIs from other machines. Again, you can use the appropriate mixin `api/public.conf`.
2. The administration API should be secured using client certificates as described in *TLS documentation section*.
3. The public API needs to be properly secured using TLS. Please follow the *corresponding instructions*.

Next Steps

The above configuration provides you with an initial setup. Without going into details, the next steps would be:

1. Configure who can join the domain by setting an appropriate permissioning strategy (default is everyone can join).
2. Configure domain parameters
3. Setup a service agreements which any client connecting has to sign before using the domain.

1.24.4.7 Multi-Node Setup

If desired, you can run many nodes in the same process. This is convenient for testing and demonstration purposes. You can either do this by listing several node configurations in the same configuration file or by invoking the Canton process with several separate configuration files (which get merged together).

```
$CANTON/bin/canton -c $CONF/storage/postgres.conf -c $CONF/nodes/domain1.conf,
↪$CONF/nodes/participant1.conf
```

1.24.5 Running in Docker

1.24.5.1 Obtaining the Docker Images

The Canton Open Source edition is published to the digitalasset/canton-open-source dockerhub repository. You can pull the Docker image using

```
docker pull digitalasset/canton-open-source[:version]
```

Here, the version is optional and by default, the latest version is used. The version `dev` is the the current main build. Please note that previous versions were called `canton-community`, before we renamed the artefact to `canton-open-source`.

If you want to use the edition included with Daml Enterprise, you can download it using

```
docker login digitalasset-canton-enterprise-docker.jfrog.io
docker pull digitalasset-canton-enterprise-docker.jfrog.io/digitalasset/canton-
↪enterprise
```

1.24.5.2 Starting Canton

The canton executable is the default image entry point so all examples using `bin/canton` can simply substitute that with `docker run digitalasset/canton`.

For example, to run with our simple topology configuration in interactive console mode:

```
docker run --rm -it digitalasset/canton-open-source:latest --config simple-
↪topology.conf
```

The `--rm` option ensures that the container is removed when the canton process exits. The `-it` options start the container interactively and provide a TTY for running our console.

The default working directory of the container is `/canton`.

By default docker will pull the `latest` tag containing the latest Canton release. As docker will only automatically pull `latest` once, ensure you have the latest version by periodically running `docker pull digitalasset/canton-open-source`.

Previous releases can be run by specifying their tag `digitalasset/canton-open-source:2.4.0`.

1.24.5.3 Configuring Logging

The default convention with logging of containers is to have the process to log to `stdout`. Therefore, we change the logging behaviour of Canton using appropriate *command line flags*, such as `--log-profile=container`.

1.24.5.4 Supplying custom configuration and DARs

To expose files to the canton container you must specify a volume mapping from the host machine to the container.

For example, if you have the local directory `my-application` containing your custom canton configuration and DAR:

```
docker run --rm -it \
   --volume "$PWD/my-application:/canton/my-application" \
   digitalasset/canton-open-source --config /canton/my-application/my-config.conf
```

DARs can be loaded using the same container local path.

1.24.5.5 Exposing the ledger-api to the host machine

Applications using Canton will typically need access to the ledger-api to read from and write to the ledger. Each participant binds the ledger-api to the port specified at the configuration key: `ledger-api.port`. For `participant1` in the simple topology example this is set to port 5011.

To expose the ledger-api to port 5011 on the host machine, run docker with the following options:

```
docker run --rm -it \
   -p 5011:5011 \
   digitalasset/canton-open-source \
   -C canton.participants.participant1.ledger-api.address=0.0.0.0 \
   --config examples/01-simple-topology/simple-topology.conf \
   --bootstrap examples/01-simple-topology/simple-ping.canton
```

The ledger-api port for each participant will need to be mapped separately.

1.24.5.6 Running Postgres in Docker

Canton requires an appropriate database to persist data. For this purpose, such a database can also be run in a docker container using the following, helpful command:

```
docker run -d --rm --name canton-postgres --shm-size=256mb --publish 5432:5432 -e
↪POSTGRES_USER=test-user
    -e POSTGRES_PASSWORD=test-password postgres:14.8-bullseye postgres -c max_
↪connections=500
```

Please note that the `--publish` command allows us to pick the target port which we have to define in the Canton configuration file. The `--rm` will delete the data store once the docker container is killed. This is useful for short-term tests. The `--shm-size 256mb` is necessary as Docker will allocate only 64mb of shared memory by default which is insufficient for the way Canton uses Postgres.

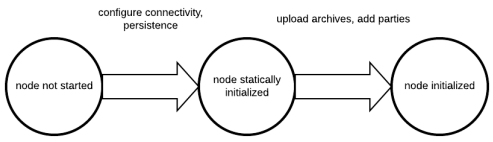
Note that you also need to create the databases yourself, which for Postgres you can do using `psql`

```
PGPASSWORD=test-password psql -h localhost -U test-user << EOF
CREATE DATABASE participant1;
GRANT ALL ON DATABASE participant1 TO CURRENT_USER;
EOF
```

The tables will be managed automatically by Canton. The `psql` solution works also if you run multiple nodes on one Postgres database which all require separate databases. If you run just one node against one database, you can avoid using `psql` by adding `--POSTGRES_DB=participant1` to above docker command.

1.24.6 Static Configuration

Canton differentiates between static and dynamic configuration. Static configuration is immutable and therefore has to be known from the beginning of the process start. An example for a static configuration are the connectivity parameters to the local persistence store or the port the admin-apis should bind to. On the other hand, connecting to a domain or adding parties however is not a static configuration and therefore is not set via the config file but through *console commands* (or the *administration APIs*).



The configuration files themselves are written in HOCON format with some extensions:

> Durations are specified scala durations using a `<length><unit>` format. Valid units are defined by scala directly, but behave as expected using `ms`, `s`, `m`, `h`, `d` to refer to milliseconds, seconds, minutes, hours and days. Durations have to be non-negative in our context.

Canton does not run one node, but any number of nodes, be it domain or participant nodes in the same process. Therefore, the root configuration allows to define several instances of domain and participant nodes together with a set of general process parameters.

A sample configuration file for two participant nodes and a single domain can be seen below.

```
canton {
  participants {
    participant1 {
      storage.type = memory
      admin-api.port = 5012
      ledger-api.port = 5011
    }
    participant2 {
      storage.type = memory
      admin-api.port = 5022
      ledger-api.port = 5021
    }
  }
  domains {
    mydomain {
      storage.type = memory
      public-api.port = 5018
      admin-api.port = 5019
    }
  }
  // enable ledger_api commands for our getting started guide
  features.enable-testing-commands = yes
}
```

1.24.6.1 Configuration reference

The Canton configuration file for static properties is based on PureConfig. PureConfig maps Scala case classes and their class structure into analogue configuration options (see e.g. the PureConfig quick start for an example). Therefore, the ultimate source of truth for all available configuration options and the configuration file syntax is given by the appropriate scaladocs of the CantonConfig classes.

When understanding the mapping from scaladocs to configuration, please keep in mind that:

> CamelCase Scala names are mapped to lowercase-with-dashes names in configuration files, e.g. `domainParameters` in the scaladocs becomes `domain-parameters` in a configuration file (dash, not underscore).
> `Option[<scala-class>]` means that the configuration can be specified but doesn't need to be, e.g. you can specify a JWT token via `token=token` in a remote participant configuration, but not specifying `token` is also valid.

1.24.6.2 Configuration Compatibility

The enterprise edition configuration files extend the community configuration. As such, any community configuration can run with an enterprise binary, whereas not every enterprise configuration file will also work with community versions.

1.24.6.3 Advanced Configurations

Configuration files can be nested and combined together. First, using the `include required` directive (with relative paths), a configuration file can include other configuration files.

```
canton {
    domains {
        include required(file("domain1.conf"))
    }
}
```

The `required` keyword will trigger an error, if the included file does not exist; without the `required` keyword, any missing files will be silently ignored. The `file` keyword instructs the configuration parser to interpret its argument as a file name; without this keyword, the parser may interpret the given name as URL or classpath resource. By using the `file` keyword, you will also get the most intuitive semantics and most stable semantics of `include`. The precise rules for resolving relative paths can be found here.

Second, by providing several configuration files, we can override configuration settings using explicit configuration option paths:

```
canton.participants.myparticipant.admin-api.port = 11234
```

If the same key is included in multiple configurations, then the last definition has highest precedence.

Furthermore, HOCON supports substituting environment variables for config values using the syntax `key = ${ENV_VAR_NAME}` or optional substitution `key = ${?ENV_VAR_NAME}`, where the key will only be set if the environment variable exists.

1.24.6.4 Configuration Mixin

Even more than multiple configuration files, we can leverage PureConfig to create shared configuration items that refer to environment variables. A handy example is the following, which allows to share database configuration settings in a setup involving several participant or domain nodes:

```
# Postgres persistence configuration mixin
#
# This file defines a shared configuration resources. You can mix it into your
↪configuration by
# refer to the shared storage resource and add the database name.
#
# Example:
#   participant1 {
#     storage = ${_shared.storage}
#     storage.config.properties.databaseName = "participant1"
#   }
#
# The user and password credentials are set to "canton" and "supersafe". As this
↪is not "supersafe", you might
# want to either change this configuration file or pass the settings in via
↪environment variables.
#
_shared {
```

(continues on next page)

(continued from previous page)

```
  storage {
    type = postgres
    config {
      dataSourceClass = "org.postgresql.ds.PGSimpleDataSource"
      properties = {
        serverName = "localhost"
        # the next line will override above "serverName" in case the environment
↪variable POSTGRES_HOST exists
        serverName = ${?POSTGRES_HOST}
        portNumber = "5432"
        portNumber = ${?POSTGRES_PORT}
        # the next line will fail configuration parsing if the POSTGRES_USER
↪environment variable is not set
        user = ${POSTGRES_USER}
        password = ${POSTGRES_PASSWORD}
      }
    }
    // If defined, will configure the number of database connections per node.
    // Please ensure that your database is setup with sufficient connections.
    // If not configured explicitly, every node will create one connection per
↪core on the host machine. This is
    // subject to change with future improvements.
    parameters.max-connections = ${?POSTGRES_NUM_CONNECTIONS}
  }
}
```

Such a definition can subsequently be referenced in the actual node definition:

```
canton {
    domains {
        mydomain {
            storage = ${_shared.storage}
            storage.config.properties.databaseName = ${CANTON_DB_NAME_DOMAIN}
        }
    }
}
```

1.24.6.5 Multiple Domains

A Canton configuration allows to define multiple domains. Also, a Canton participant can connect to multiple domains. This is however only supported as a preview feature and not yet suitable for production use.

In particular, contract key uniqueness cannot be enforced over multiple domains. In this situation, we need to turn contract key uniqueness off by setting

```
canton {
    domains {
        alpha {
            // subsequent changes have no effect and the mode of a node can never
↪be changed
            init.domain-parameters.unique-contract-keys = false
        }
    }
```

(continues on next page)

(continued from previous page)

```
    participants {
        participant1 {
            // subsequent changes have no effect and the mode of a node can never
↪be changed
            init.parameters.unique-contract-keys = false
        }
    }
}
```

Please note that the setting is final and cannot be changed subsequently. We will provide a migration path once multi-domain is fully implemented.

1.24.6.6 Fail Fast Mode

Be default, Canton will fail to start if it cannot access some external dependency such as the database. This is preferable during initial deployment and development, as it provides instantaneous feedback, but can cause problems in production. As an example, if Canton is started with a database in parallel, the Canton process would fail if the database is not ready before the Canton process attempts to access it. To avoid this problem, you can configure a node to wait indefinitely for an external dependency such as a database to start. The config option below will disable the fail fast behaviour for `participant1`.

```
canton.participants.participant1.storage.parameters.fail-fast-on-startup = "no"
```

This option should be used with care as, by design, it can cause infinite, noisy waits.

1.24.6.7 Init Configuration

Some configuration values are only used during the first initialization of a node and cannot be changed afterwards. These values are located under the *init* section of the relevant configuration of the node. Below is an example with some init values for a participant config

```
participant1 {
  init {
    // example settings
    ledger-api.max-deduplication-duration = 1 minute
    parameters.unique-contract-keys = false
    identity.node-identifier.type = random
  }
}
```

The option `ledger-api.max-deduplication-duration` sets the maximum deduplication duration that the participant's *ledger configuration service* reports and uses for *command deduplication*.

Important: This feature is only available in Canton Enterprise

1.24.7 Enterprise Drivers

The Daml Enterprise edition of the Canton ledger provides the following drivers in addition to the PostgreSQL-based domain in the open-source edition.

Important: This feature is only available in Canton Enterprise

1.24.7.1 Oracle Domain

In Daml Enterprise, you can run a Canton domain (sequencer, mediator, domain manager nodes) based on an Oracle enterprise database.

Refer to the following section on how to setup Oracle for Canton as well as how to configure Canton to use an Oracle database.

> *Installation and Configuration of Canton on Oracle*

Important: This feature is only available in Canton Enterprise

1.24.7.2 Fabric Domain

The Canton-on-Fabric integration runs a Canton domain where events are sequenced using the Hyperledger Fabric ledger.

Tutorial

To run the demo Canton Fabric deployment, you will need access to the following:

> a Daml Enterprise release with drivers for access to the example files and the Canton binary
> *Canton Enterprise docker repository* access, in order to obtain the Canton docker image

Also make sure to have docker and docker-compose installed.

The following example explains how to set up Canton on Fabric using a topology with 2 sequencer nodes, (belonging to two different organizations) a domain manager, a mediator, and two participants nodes.

The demo can be found in the examples directory of the Canton Enterprise release. Unpack the Canton Enterprise release and then `cd` into `examples/e01-fabric-domain/canton-on-fabric`.

Run the script `./run.sh full`.

The script will start the following:

1. A Fabric ledger with 2 peers and one orderer node.
2. Two Canton Sequencer nodes that interact with the Fabric ledger.
3. A Canton process running a Canton domain manager, a mediator, and 2 participants. The configuration for this Canton process is in `config/canton/demo.conf`

Once the script has finished setting up (you should see the `canton` service print Successfully initialized Canton-on-Fabric together with the Canton console startup message), you will be able to interact with the two participants using the config at `config/remote/demo.conf`.

You can start an instance of the Canton console to connect to the two remote participants (provided you have also installed Canton):

```
<<canton-release>>/examples/e01-fabric-domain/canton-on-fabric$ ../../../bin/
↪canton -c config/remote/demo.conf
```

You can then perform various commands in the Canton console:

```
@ remoteParticipant1.id
res1: ParticipantId = PAR::participant1::012c7af9...

@ remoteParticipant1.domains.list_connected
res2: Seq[(com.digitalasset.canton.DomainAlias, com.digitalasset.canton.
↪DomainId)] = List((Domain 'myDomain', myDomain::01dafa04...))

@ remoteParticipant1.health.ping(remoteParticipant2)
res3: concurrent.duration.Duration = 946 milliseconds
```

User Manual

The example files located at `examples/e01-fabric-domain/canton-on-fabric` provide you with more flexibility than to run the basic demo just shown.

You will find in this directory our main script called `run.sh`. If you run the script, it will show you the help instructions with all the options that you can choose to run the deployment with.

The demo deployment will by default use the Canton version from the release. If you wish to use a different version, you can specify it with the `CANTON_VERSION` environment variable. For example, `export CANTON_VERSION=2.5.1` to use Canton v2.5.1. You can choose `dev` for the latest main build of Canton.

Depending on which options you choose, it will run a docker-compose command using a different subset of the following docker-compose files below:

> `docker-compose-ledger.yaml`: Sets up the Fabric ledger. You can see that there is a service in it called `ledger-setup` that is a service responsible for creating the crypto materials, setting up the channel and deploying the chaincode. It uses a customized and simplified version of the `test-network` from fabric-samples inside a docker container.
> `docker-compose-blockchain-explorer.yaml`: Runs a blockchain explorer that allows visualizing the Fabric ledger on the browser.
> `docker-compose-canton.yaml`: Runs all canton components: a domain manager, a mediator, the two Fabric sequencer(s) and two participants.

The bootstrapping process of the distributed domain is done by the `docker-compose-canton.yaml` docker-compose file which uses the `config/canton/demo.canton` script. If you wish to learn more about this process please refer to *domain bootstrapping*.

Run with Docker Compose

The script `run.sh` works by running `docker-compose` using a different combination of the `docker-compose` files shown above, depending on the arguments given to the script.

As was shown, to run Canton with two Fabric Sequencers in a multi-sequencer setup, run `./run.sh full`. That is equivalent to running the following docker-compose command:

```
<<canton-release>>/examples/e01-fabric-domain/canton-on-fabric$ COMPOSE_PROJECT_
↪NAME="fabric-sequencer-demo" docker-compose -f docker-compose-ledger.yaml -f
↪docker-compose-canton.yaml up
```

Note that you can at this point connect the remote participants to this setup just like in demo from the tutorial.

Cleanup

When you're done running the sequencer, make sure to run `./run.sh down`. This will clean up all docker resources so that the next run can happen smoothly.

Using the Canton Binary instead of docker

To run the full Canton setup separately outside of docker (with the canton binary or jar):

```
<<canton-release>>/examples/e01-fabric-domain/canton-on-fabric$ ./run.sh ledger
```

After a few seconds you should see the two peers and one orderer nodes are up by running `docker ps` and seeing two `hyperledger/fabric-peer` containers exposing ports 9051 and 7051 and one `hyperledger/fabric-orderer` exposing the port 7050. Next run the following:

```
<<canton-release>>/examples/e01-fabric-domain/canton-on-fabric$ ../../../bin/
↪canton -c config/self-contained/demo.conf --bootstrap config/canton/demo.canton
```

To run the jar file instead of the canton binary, simply replace `../../../bin/canton` above with `java -jar ../../../lib/canton-enterprise-*.jar`.

Blockchain Explorer

If you wish to start the Hyperledger Blockchain Explorer to browse activity on the running Fabric Ledger, add the `-e` flag when running `./run.sh`.

Alternatively you can use docker-compose as shown before and add `-f docker-compose-blockchain-explorer.yaml`.

You will then be able to see the explorer web UI in your browser if you go to `http://localhost:8080`.

You can start the explorer separately after the ledger has been started by simply running the following command:

```
<<canton-release>>/examples/e01-fabric-domain/canton-on-fabric$ COMPOSE_PROJECT_
↪NAME="fabric-sequencer-demo" docker-compose -f docker-compose-blockchain-
↪explorer.yaml up
```

(continues on next page)

(continued from previous page)

Note that even when the explorer is working perfectly, it might output some error messages like the following which can be safely ignored:

```
[ERROR] FabricGateway - Failed to get block 0 from channel undefined : □
↪TypeError: Cannot read property 'toString' of undefined
```

Fabric Setup

The Fabric Sequencer operates on top of the Fabric Ledger and uses it as the source of truth for the state of the sequencer (all the messages and the order of them).

In order for The Fabric Sequencer to successfully operate on a given Fabric Ledger, that ledger must have been set up with at least one channel where the Canton Sequencer chaincode has been installed and the sequencer needs to be configured properly to have access to the ledger.

As mentioned previously, for our demo setup we use a slightly modified version of the `test-network` scripts from fabric-samples inside a docker container to setup a simple local docker-based Fabric network. This script uses many of the Fabric CLI commands to set up this network, such as configtxgen, peer channel, peer chaincode, and peer lifecycle. In a real-life scenario one might use this CLI to set up the ledger or some specific UI provided by a cloud service provider that hosts Blockchain services.

Regarding the chaincode setup, the Fabric Sequencer expects that the chaincode is initialized by calling the function `init` (no arguments needed) and with the `--isInit` flag turned on. You can find the chaincode source at `/ledger-setup/chaincode/src/github.com/digital-asset/sequencer`.

In order to configure a Fabric Sequencer in Canton, make sure to set `canton.sequencers.<your sequencer>.sequencer.type = "fabric"`. The rest of the Fabric sequencer-specific config will be under `canton.sequencers.<your sequencer>.sequencer.config`. Within this subconfig, you'll need to set the `user` key with Fabric client details so that the sequencer can invoke chaincode functions and read from the ledger. You'll also need to set `organizations` details which include peers and orderers connection details that the sequencer will have access to. You must define at least one peer that is from the same organization as your user.

The sequencer needs access to at least enough peers to fulfil the chaincode endorsement policy. An endorsement policy that requires a single peer is enough and is what we recommend (more at *Endorsement Policies*).

It is possible to indicate the channel name with the `channel.name` key and the chaincode name with the `channel.chaincode.name` key (defaults to sequencer). This is all exemplified, including extensive commentary, in the config file used for the first sequencer of the demo, which you can find at `examples/e01-fabric-domain/canton-on-fabric/config/fabric/fabric-config-1.conf`.

By default, the sequencer application will start reading blocks from the ledger from the genesis block. We can signal a later starting point by setting `channel.chaincode.start-block-height` to a specific number in case the chaincode has been deployed much later than genesis.

Block Cutting Parameters and Performance

It is possible to configure the block cutting parameters of the ledger by changing the file at `ledger-setup/configtx/configtx.yaml`.

The relevant parameters are the following:

> `Orderer.BatchTimeout`: The amount of time to wait before creating a block.
> `Orderer.BatchSize.MaxMessageCount`: The maximum number of transactions to permit in a block (block size).

Note: In other kinds of Fabric Ledger setups, one should be able to configure these parameters in different ways.

If your use case operates under high traffic, you may benefit from increasing the block size in order to increase your throughput at the expense of latency. If you care more about latency and don't need to support high traffic, then decreasing block size will be of help.

Currently, we have set the values of 200ms for batch timeout and 50 for block size as it has empirically shown to be a good tradeoff after some rounds of long running tests, but feel free to pick parameters that fit your use-case best.

Note: See slide 17 of http://www.mscs.mu.edu/~mascots/Papers/blockchain.pdf for a discussion on block size influence on throughput and latency.

Authorization

When operating the Fabric infrastructure to support the Fabric Sequencer one may want to authorize only certain organizations to determine the sequencer's behavior.

Only the organizations included in the Fabric Channel will be able to operate on the ledger.

Fabric Policies can also be used to limit how the capabilities of organizations in the channel. See more on that under *Endorsement Policies* below.

Endorsement Policies

Fabric Policies can be used to define how members come to agreement on accepting or rejecting changes to the network, a channel or a smart contract.

Versatile policies can be written using combinations of `AND`, `OR` and `NOutOf` (more detail here).

The most relevant kinds of policies for our purposes here are the channel configuration policy (defined at the channel level) and endorsement policies (defined at the chaincode level).

See other kinds of policies here.

We recommend setting up a single peer endorsement policy.

We do not benefit from the chaincode endorsements because there is no mutable state in the chaincode or special logic that needs to be endorsed. We care more about correct ordering of blocks, which is taken care of by the ordering service. Because of that, there is no point in using more complex endorsement policies. A single peer endorsement policy also simplify configuration and increases availability. The demo we ship is configured like this (at *ledger-setup/configtx/configtx.yaml*, under *Application.Policies.Endorsement.*)

High Availability

When configuring the Fabric sequencer, make sure to provide access to at least enough peers to fulfill the chaincode endorsement policy that has been configured.

Access to additional peers may also be configured, to make the setup more highly available and to avoid a scenario where the crash of one peer would cause transactions to stop going through due to lack of enough endorsements.

If a client is connected to more than one Fabric Sequencer and each sequencer defines a different set of connections to Fabric peers (and orderers), the client will benefit from another level of availability. If of the sequencers is not healthy, the client will simply fail over to the ones that are still healthy.

Important: This feature is only available in Canton Enterprise

---

1.24.7.3 Ethereum Domain

Introduction

Daml Enterprise includes a Canton Ethereum Sequencer integration, which interacts via an Ethereum client with a smart contract `Sequencer.sol` deployed on an external Ethereum network. It uses the blockchain as source-of-truth for sequenced events and is currently tested with the Ethereum client Hyperledger Besu. The *architecture document* contains more details on the architecture of the integration.

The Ethereum Demo

Prerequisites

To run the demo, you will need access to a Daml Enterprise release with drivers, the *Canton docker repository*, as well as having docker, docker-compose, and Hyperledger Besu (instructions here) installed.

Introduction

The demo Ethereum deployment can be found inside the examples directory of the Daml Enterprise release with drivers. Unpack the Canton Drivers release and then `cd` into `examples/e03-ethereum-sequencer`.

The script `./run.sh` from the folder `examples` will create a new Besu testnet for the demo deployment and then start the demo. It has two scenarios: a simple and an advanced scenario. Both scenarios will start several dockerised services:

An ethereum testnet, using four Besu nodes with the QBFT consensus protocol. This is the same for the simple and advanced scenario.

An instance of Canton. This includes two Participants and a Domain with one Ethereum sequencer for the simple scenario and two Ethereum sequencers for the advanced scenario. The respective Canton configurations are in `canton-conf/simple` and `canton-conf/advanced`.

The environment variable `CANTON_VERSION` is used to select the version of Canton to use for the demo deployment. This should normally be set to the version of Canton being used, but can alternatively be set to a different version or `dev` for the latest main build of Canton.

Simple Scenario

The simple scenario uses one Canton sequencer whose corresponding `Sequencer.sol` contract is deployed using a script before startup. It uses mutual TLS between Canton and Besu.

Advanced Scenario

The advanced scenario uses two Canton sequencers, mutual TLS, Ethereum wallets and uses the same script deploying `Sequencer.sol`.

Running a scenario

To start the simple or advanced demo scenario run:

```
<<canton-driver-release>>/examples/e03-ethereum-sequencer$ CANTON_VERSION=<your
↪version> ./run.sh simple
```

or

```
<<canton-driver-release>>/examples/e03-ethereum-sequencer$ CANTON_VERSION=<your
↪version> ./run.sh advanced
```

A new Besu testnet will be created and the demo will begin running with the created testnet. Once the demo is initialized and running, it will print out

```
*****************************
Successfully initialized Canton-on-Ethereum
*****************************
```

You will then be able to interact with the two participants via their ledger APIs (or their admin APIs) respectively running on ports `5011` and `5021` (or `5012` and `5022`).

For example, you can start an instance of the Canton console to connect to the two remote participants. You can find the Canton binary in `bin/canton` of the Canton Enterprise release artifact.

```
<<canton-driver-release>>/examples/e03-ethereum-sequencer$ ../../bin/canton -c
↪canton-conf/remote.conf
```

You can then perform various commands in the Canton console:

```
@ remoteParticipant1.id
res5: ParticipantId = ParticipantId(
  UniqueIdentifier(Identifier("participant1"), Namespace(Fingerprint(
↪"01e69a39e2c821fc98eaeb22994b47084162122a01ebcb16dfb2514ccafcedd43d")))
)

@ remoteParticipant2.id
res6: ParticipantId = ParticipantId(
  UniqueIdentifier(Identifier("participant2"), Namespace(Fingerprint(
↪"014aeb29dddff83678bc6f1194c363c6f0d18d3a6c9655927a7fb5adc84ec0532c")))
)

@ remoteParticipant1.domains.list_connected
res7: Seq[(com.digitalasset.canton.DomainAlias, com.digitalasset.canton.
↪DomainId)] = List(
  (Domain 'mydomain', mydomain::01537eb8...)
)

@ remoteParticipant1.health.ping(remoteParticipant2)
res8: concurrent.duration.Duration = 968 milliseconds
```

To shutdown and remove all Docker containers, you can execute `stop-with-purge.sh`:

```
<<canton-driver-release>>/examples/e03-ethereum-sequencer$ ./stop-with-purge.sh
```

Generating a Clean Testnet

The directory `examples/e03-ethereum-sequencer/qbft-testnet` contains the script `generate-testnet.sh`. This automatically generates a clean Besu network in a `testnet` directory, including new randomized private keys. `generate-testnet.sh` is automatically called by `run.sh` but you may want to understand and edit it to create your own custom Besu deployment.

When `generate-testnet.sh` is run:

> The state from any previous runs of `generate-testnet.sh` is deleted and a new directory `testnet` is created.
> A genesis file, a set of keys for four Besu nodes and TLS certificates for Canton and Besu are automatically generated. These can be found in the folders `testnet/nodei` (where i has values 1 to 4) and `testnet/tls`. respectively.
> The four Besu nodes are started via calling `start-node.sh`.

If the script finds Besu keys or TLS certificates in the same directory as the script, it will attempt to reuse them. This significantly reduces startup time if you want to test different network configurations.

The generated Besu testnet has been configured largely following these tutorials:

> https://besu.hyperledger.org/en/stable/private-networks/tutorials/qbft/
> and https://besu.hyperledger.org/en/stable/HowTo/Configure/FreeGas/

Note that the RPC HTTP APIs `ETH`, `TXPOOL`, and `WEB3` of Besu need to be enabled when using the Besu driver.

Customization of the Besu network

The parameters of the generated testnet can be changed by modifying the `genesis.json` file defined inline in `generate-testnet.sh`. Similarly, the CLI options with which the Besu nodes are started can be configured by modifying `start-node.sh`

Customization of the Demo Configuration

You can also modify the Canton configurations and bootstrap scripts for the demo if, for example, you want to *add persistence to the participants*. The Canton configurations are found in

`canton-conf/simple` and
`canton-conf/advanced`

for the simple and advanced scenarios, respectively. If you want to change Ethereum-specific configuration options, (e.g. to configure a different wallet) please refer to the documentation section on this page and the corresponding scaladoc configuration option.

Note that if you change port mappings in the Canton config file you may also need to update the corresponding docker compose files in directory `docker-compose/`.

Error codes

The Ethereum Sequencer application auto-detects many common configuration and deployment issues and logs them as warnings or errors with *error codes*. If you see such a warning or error, please refer to the *respective error code explanation and resolution*.

TLS configuration

Canton supports mutual TLS between Canton and Ethereum client nodes and the demo contains an example of how to configure this. Concretely, the TLS configuration for Canton expects a key store and the path to the Ethereum TLS certificates:

```
_tls {
  canton-key-store {
    path="/canton/testnet-working/tls/canton_store.p12"
    password="password"
  }
  ethereum-certificate-path = "/canton/testnet-working/tls/besu_cert.pem"
}

canton.sequencers.ethereumSequencer1.sequencer.config.tls = ${_tls}
```

The demo also contains the utility script `qbft-testnet/generate-tls.sh` which is called by `generate-testnet.sh` and writes the TLS certificates to `qbft-testnet/testnet/tls`. These certificates are then used by `start-node.sh`.

If Canton is not configured to use TLS with an Ethereum node, it will attempt to communicate via a HTTP endpoint on the Ethereum node (and HTTPS for TLS).

For more details on the Canton configuration, please see the scaladocs of the TLS configuration. For more details on how to configure Besu to accept TLS connections (as done in the demo, see especially file `start-node.sh`), please see the Besu documentation.

Ethereum accounts and wallets

Canton allows you to configure an Ethereum wallet (and therefore an Ethereum account) to be used by an Ethereum sequencer application. The configured Ethereum account is used for all interactions of the Ethereum sequencer with the Ethereum blockchain. If no Ethereum account is explicitly configured, a random Ethereum account is used.

Note: When multiple Ethereum sequencer applications interact with the same `Sequencer.sol` instance, each Ethereum Sequencer process needs to use a separate Ethereum account. Otherwise, transactions may get stuck due to nonce mismatches.

Canton allows configuring a wallet in UTC JSON and BIP 39 format.

The Ethereum demo includes examples of mix-in wallet configuration files for both formats; the UTC JSON-based wallet mix-in looks as follows:

```
canton.sequencers.ethereumSequencer2.sequencer.config.wallet {
  type = "utc-json-wallet"
  password = "password"
  wallet-path = "advanced/utc-wallet.json"
}
```

with following `utc-wallet.json`:

```
canton.sequencers.ethereumSequencer2.sequencer.config.wallet {
  type = "utc-json-wallet"
  password = "password"
  wallet-path = "advanced/utc-wallet.json"
}
```

The BIP39-based wallet mix-in looks as follows:

```
canton.sequencers.ethereumSequencer2.sequencer.config.wallet {
  type = "utc-json-wallet"
  password = "password"
  wallet-path = "advanced/utc-wallet.json"
}
```

For more details, please refer to the Canton scaladoc documentation.

Deployment of the sequencer contract

Manual deployment

If you want to manually deploy `Sequencer.sol` to your Ethereum network, the file `<<canton-drivers-release>>/examples/e03-ethereum-sequencer/qbft-testnet/Sequencer.bin-runtime` contains the compiled Solidity code you need to deploy. For Besu, for example, you will need to specify the contents of `Sequencer.bin-runtime` in `"code": "..."` as documented here. This can also be seen in the `generate-testnet.sh` script.

When a contract is deployed using the `genesis.json`, the constructor is never called. Therefore, any variables initialized as part of the constructor need to be set using the `"storage": "..."` configuration block.

```
"alloc": {
     "0x0af0238112db255e1a2c8a6c1cd4e122c56bbc38": {
           "code": "'"$contractCode"'",
           "storage": {
            "4":
↪"0x312e302e310000000000000000000000000000000000000000000000000000000a"
           }
     }
```

The above `alloc` block from the `genesis.json` deploys the contract code and initializes any fields needed.

Requirements for the Ethereum Network

The Canton Ethereum integration is currently tested with the QBFT consensus protocol as illustrated in the demo. Other setups are possible, but they should fulfill the following requirements:

- The Ethereum client Hyperledger Besu (version in release notes tested, but any sibling sub-minor version should be OK) should be used and expose the RPC HTTP APIs `ETH`, `TXPOOL`, and `WEB3`.
- Currently, a free gas network is required. This means setting the gas price to zero.
- The block size limit (often measured in gas, and sometimes referred to as the 'gas limit') must be larger than any message to be sequenced. It is recommended to set this parameter as high as possible.
- The contract size limit must be big enough for the Canton Ethereum Domain to store all required state for sequencing messages. It is recommended to set this parameter as high as possible.
- Proof of authority protocols are recommended over proof of work.
- Currently, consensus protocols must have immediate finality. This means that ledger forks should not occur with the chosen consensus protocol.

Furthermore, we also have some suggestions to improve throughput and latency irrespective of the choice of Ethereum client.

Throughput

Generally, the throughput of a Canton system using Ethereum-based sequencers is limited by the throughput of the Ethereum client. Thus, if an Ethereum-based sequencer does not deliver the desired throughput, the throughput and deployment of the Ethereum clients should be optimized in the first instance. For Besu performance optimization, some recommendations can be found in the Besu documentation - in particular, it is crucial to use a fast storage media.

Latency

Within a Canton transaction, there are three sequential sequencing steps, that is, a single Canton transaction leads to at least three sequential messages sent to the sequencer. This is illustrated, e.g., in the *message sequence diagram* of the Canton 101 section. As a result, a Canton transaction also leads to at least three Ethereum transactions within three different blocks. Thus, to achieve relatively low latencies, the Ethereum network networks must be configured with a frequent block mining frequency (configured via `blockperiodseconds` in Besu) and ideally co-located with the Canton sequencer node. A block mining frequency of at least one block per second is recommended.

Trust Properties of the Ethereum Sequencer Integration

The demo integration uses two participants and two different Ethereum Sequencer nodes. Each participant chooses its preferred Ethereum Sequencer node, and this node performs reads and writes on behalf of the participant. Therefore, each participant must trust its chosen Ethereum Sequencer node. Additionally, each participant must trust some proportion of the nodes in the Ethereum network as determined by the consensus protocol.

High Availability

The Ethereum sequencer currently supports connecting to just one Ethereum client node. The sequencer node monitors its dependencies and signals to its users any potential issue that would prevent it from operating correctly.

The health information is exposed as a Grpc Health service; sequencer clients use this in order to determine whether a sequencer is usable or or not. The health state is also included in the sequencer status accessible on the Admin API. In order to benefit from higher availability, clients must connect to multiple sequencers such that they can fail over automatically to healthy sequencers once some of them become unhealthy.

The following health checks are implemented:

> Can the sequencer node connect and read from the Ethereum RPC API by calling the eth_synching method and check whether a result can be obtained?
> Can the sequencer node connect to its database?

1.25 High Availability (HA)

The following sections cover how High Availability (HA) is implemented in Daml solutions.

1.25.1 Intro to HA in Canton

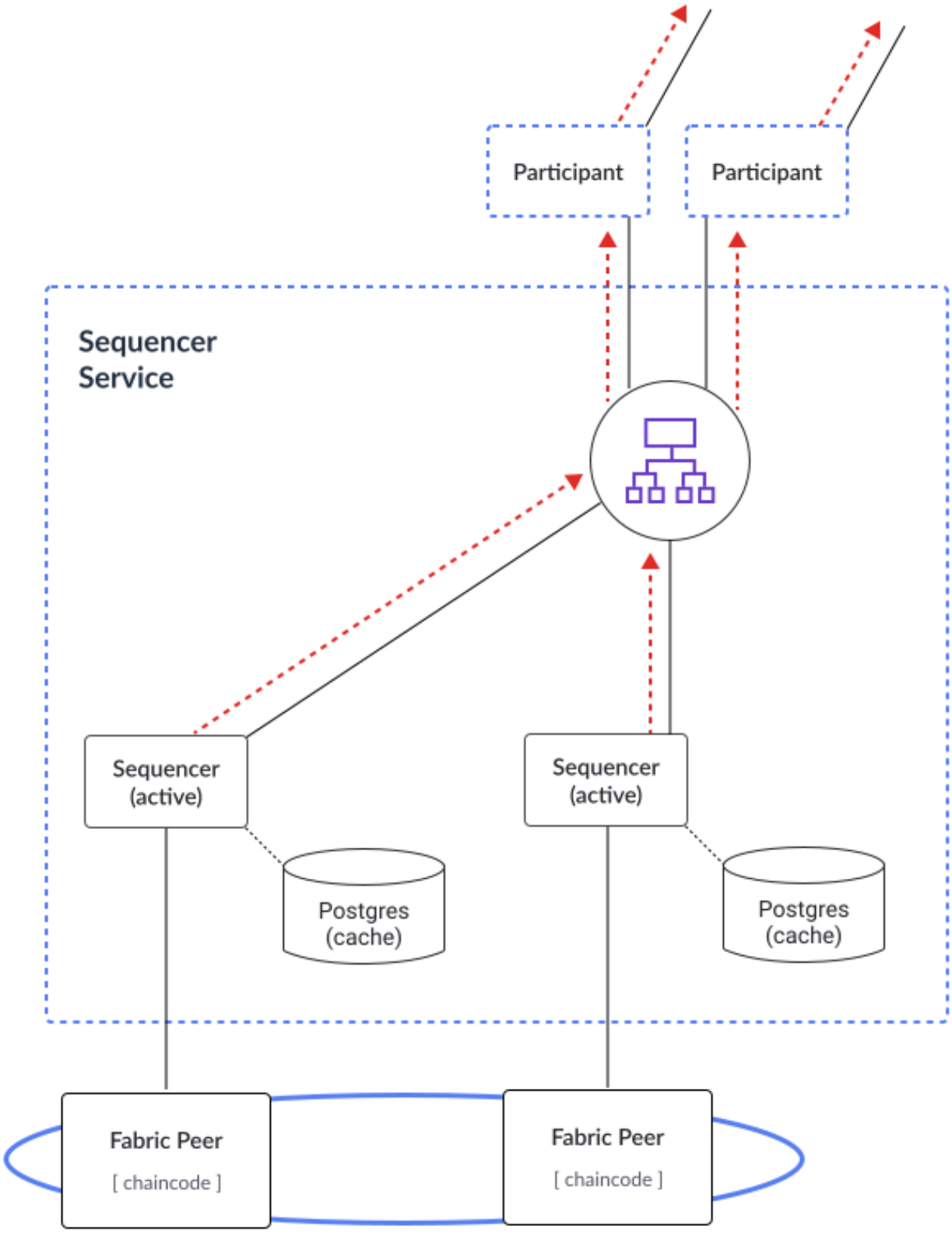
This section provides an overview of High Availability (HA) in Canton, how participant nodes are replicated, and how HA is implemented on the domain.

1.25.1.1 Overview

High Availability (HA) is the elimination of single points of failure to ensure that applications continue to operate when a component they depend on, such as a server, fails.

HA for Daml solutions focuses on the following components running in separate processes:

Participant nodes
Domains:
- Sequencer
- Mediator
- Domain Topology Manager


Participant
Participant
Sequencer
Service
Sequencer
(active)
Sequencer
(active)
Postgres
(cache)
Postgres
(cache)
Fabric Peer
[chaincode]
Fabric Peer
[chaincode]

Participant Nodes

The availability of a participant node shouldn't affect the availability of another participant node, except for the following workflows:

1. Where they are both involved.
2. When they have distinct visibility configurations, i.e. they manage different parties involved in the workflow.

For example, if they both host the same party, transactions involving the party can continue as long as either of them is available.

Note: An application operating on behalf of a party cannot transparently failover from one participant node to another due to the difference in offsets emitted on each participant.

Domains

A participant node's availability is not affected by the availability of the domain, except for workflows that use the domain. This allows participant nodes and domains to take care of their HA separately.

Replication

To achieve HA, components replicate. All replicas of the same component have the same trust assumptions, i.e. the operators of one replica must trust the operators of the other replicas.

Databases

In general, when a component is backed by a database/ledger, the component's HA relies on the HA of the database/ledger. Therefore, the component's operator must handle the HA of the database separately.

All database-backed components are designed to be tolerant of temporary database outages. During the database failover period, components halt processing until the database becomes available again, resuming thereafter.

Transactions that involve these components may time out if the failover takes too long. Nevertheless, they can be safely resubmitted, as command deduplication is idempotent.

Health Check

Canton components expose a *health endpoint*, for checking the health of the components and their subcomponents.

Important: This feature is only available in Canton Enterprise

---

1.25.1.2 Replicating Participant Nodes

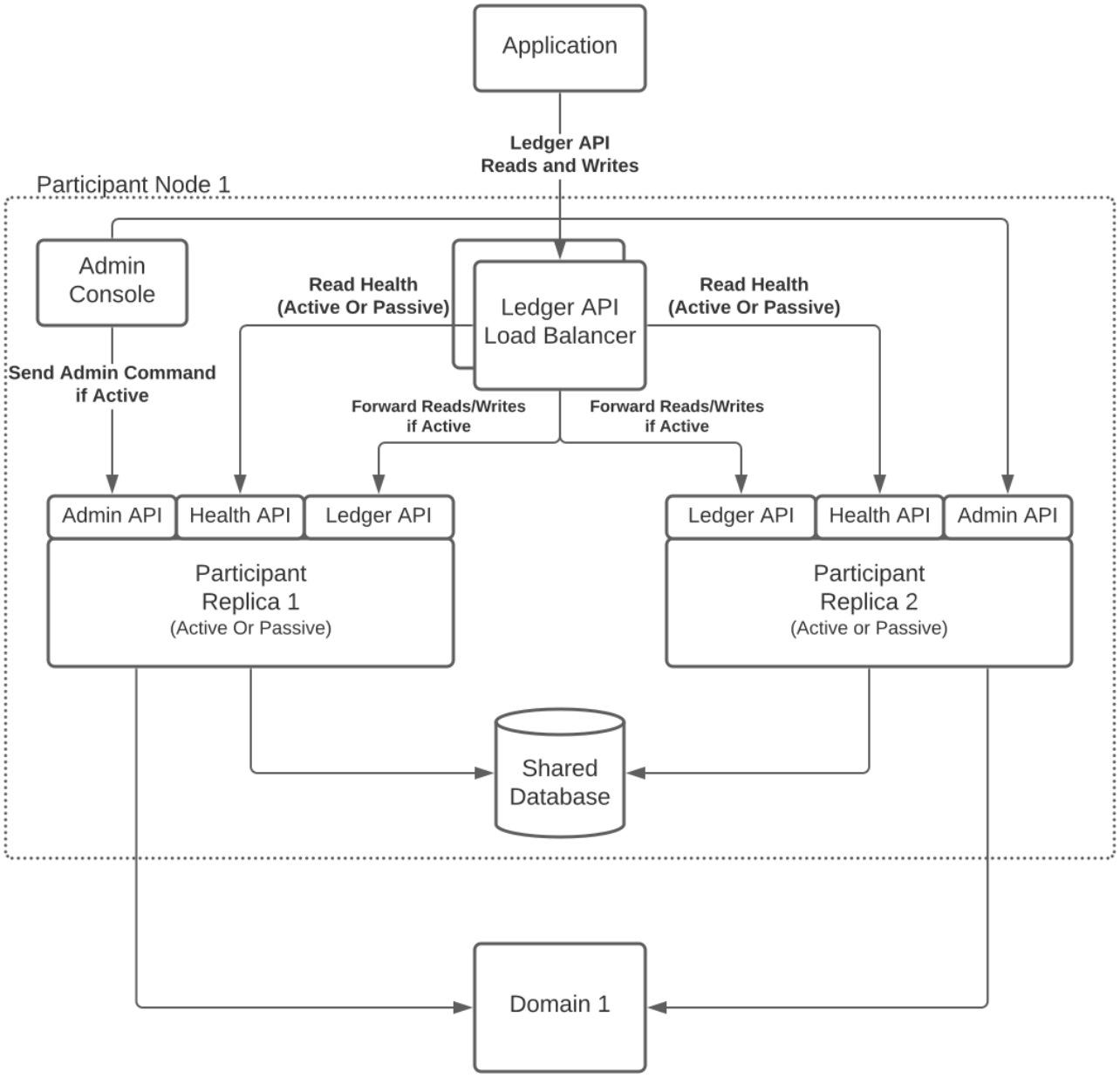
Participant nodes are replicated in an active-passive configuration with a shared database.

The active node services requests while one or more passive replicas wait in warm-standby mode, ready to take over if the active replica fails.

High-Level System Design

A logical participant node - shown below - contains multiple physical participant node replicas, all using a shared database.

Each replica exposes its own Ledger API although these can be hidden by a single Ledger API endpoint running on a highly available load balancer.



The load balancer configuration contains details of all Ledger API server addresses and the ports for

the participant node replicas. Replicas expose their active or passive status via a health endpoint. The load balancer can also detect when the backend port becomes unreachable, i.e. when the ledger API server is shut down as a node goes from active to passive.

Periodically polling the health API endpoint, the load balancer identifies a replica as offline if it is passive. Requests are then *only* sent to the active participant node.

Important: The health endpoint polling frequency can affect the failover duration.

During failover, requests may still go to the former active replica; which rejects them. The application retries until the requests are forwarded to the new active replica.

Shared Database

The replicas require a shared database for the following reasons:

1. To share the command ID deduplication state of the Ledger API command submission service. This prevents double submission of commands in case of failover.
2. To obtain consistent ledger offsets without which the application cannot seamlessly failover to another replica. The database stores ledger offsets in a non-deterministic manner based on the insertion order of publishing events in the multi-domain event log.

Leader Election

A leader election establishes the active replica. The participant node sets the chosen active replica as the single writer to the shared database.

Exclusive, application-level database locks - tied to the database connection lifetime - enforce the leader election and set the chosen replica as the single writer.

Note: Alternative approaches for leader election, such as Raft, are unsuitable because the leader status can be lost between the leader check and the use of the shared resource, i.e. writing to the database. Therefore, we cannot guarantee a single writer.

Exclusive Lock Acquisition

A participant node replica uses a write connection pool that is tied to an exclusive lock on a main connection, and a shared lock on all pool connections. If the main lock is lost, the pool's connections are ramped down. The new active replica must wait until all the passive node's pool connections are closed, which is done by trying to acquire the shared lock in exclusive mode.

Note: Using the same connection for writes ensures that the lock is active while writes are performed.

Lock ID Allocation

Exclusive application-level locks are identified by a 30-bit integer lock id which is allocated based on a scope name and counter.

The lock counter differentiates locks used in Canton from each other, depending on their usage. The scope name ensures the uniqueness of the lock id for a given lock counter. The allocation process generates a unique lock id by hashing and truncating the scope and counter to 30 bits.

Note: On Oracle, the lock scope is the schema name, i.e. user name. On PostgreSQL, it is the name of the database.

Participant replicas must allocate lock ids and counters consistently. It is, therefore, crucial that replicas are configured with the same storage configuration, e.g. for Oracle using the correct user-name to allocate the lock ids within the correct scope.

Prevent Passive Replica Activity

Important: Passive replicas do not hold the exclusive lock and cannot write to the shared database.

To avoid passive replicas attempting to write to the database - any such attempt fails and produces an error - we use a coarse-grained guard on domain connectivity and API services.

To prevent the passive replica from processing domain events, and ensure it rejects incoming Ledger API requests, we keep the passive replica disconnected from the domains as coarse-grained enforcement.

Lock Loss and Failover

If the active replica crashes or loses connection to the database, the lock is released and a passive replica can claim the lock and become active. Any pending writes in the formerly active replica fail due to losing the underlying connection and the corresponding lock.

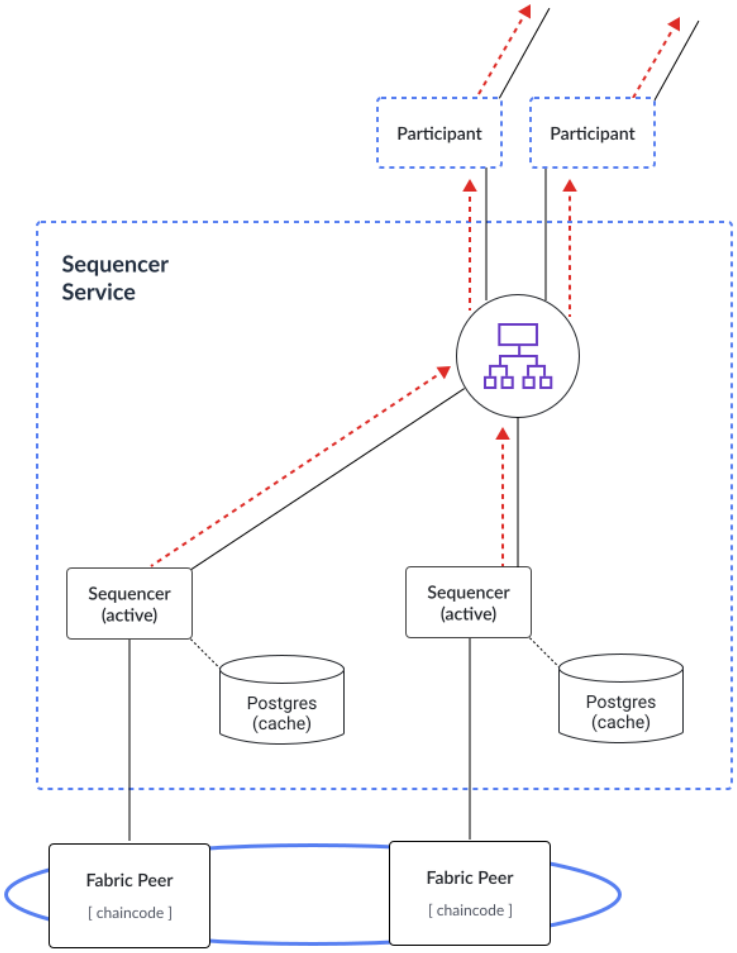
The active replica has a grace period in which it may rebuild the connection and reclaim the lock, due to the higher frequency of health checks on the lock in the active replica vs. the passive replica trying to acquire the lock at a lower frequency.

The passive replicas continuously attempt to acquire the lock within a configurable interval. Once the lock is acquired, the participant replication manager sets the state of the successful replica to active.

When a passive replica becomes active, it connects to previously connected domains to resume event processing. The new active replica accepts incoming requests, e.g. on the Ledger API which starts when the node becomes active. The former active replica, which is now passive, shuts down its Ledger API to stop accepting incoming requests.

1.25.1.3 HA on the Domain

The diagram shows a domain containing the topology manager, mediator, and sequencer components.



A domain is fully available only when all components are available. However, transaction processing still runs even when only the mediator and the sequencer are available.

As all of the domain components run in separate processes, HA is architected per component. This means that in an HA deployment, the domain is not deployed as a domain node (which would run the mediator, sequencer, and manager in a single process).

Sequencer

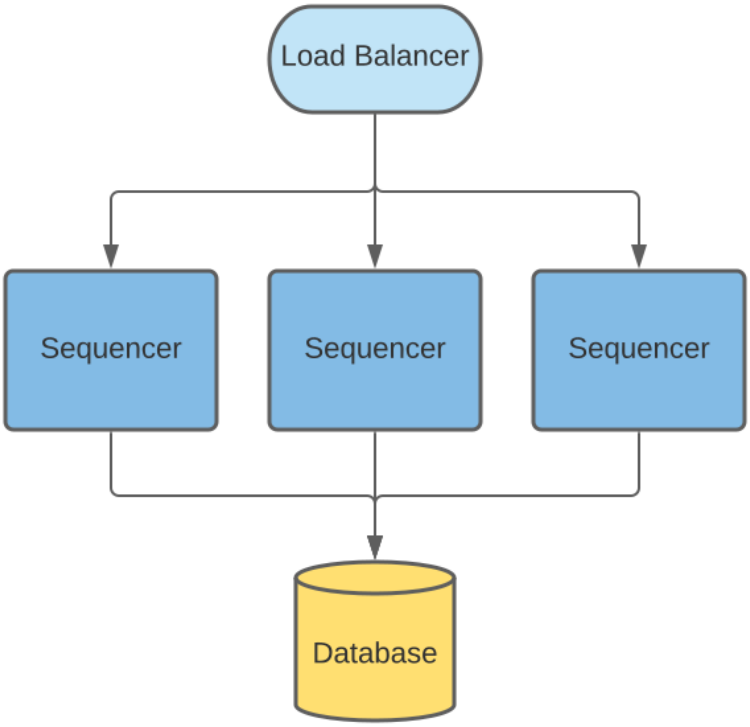
Sequencer HA depends on the chosen implementation. For example, when using a ledger such as Hyperledger Fabric, HA is already set up. Multiple sequencer nodes must be deployed.

The domain returns multiple sequenced endpoints, any of which can be used to interact with the underlying ledger.

Database Sequencer

The database sequencer has an active-active setup over a shared database.

The sequencer relies on the database for both HA and consistency. The database ensures that events are sequenced in a consistent order.



Many sequencer nodes can be deployed. Each node has concurrent read and write components when accessing the database. The load balancer evenly distributes requests between sequencer nodes.

Note: The system stops sending requests to an unhealthy node.

Consistency and the Database Sequencer

Each node is assigned a distinct index from the total number of sequencer nodes. The index is included in event timestamps to ensure that sequencer nodes never use duplicate event IDs/timestamps.

Events are written to the `events` table in ascending timestamp order. Readers need to know the point at which events can be read without the risk of an earlier event being inserted by a write process. To do this, writers regularly update a `watermark` table into which they publish their latest event timestamp. Readers take the minimum timestamp from the table as the point from which they can safely query events.

Failing Sequencer Nodes and the Database Sequencer

If a sequencer node fails, it stops updating its `watermark` value and, when the value reaches the minimum timestamp, all readers pause as they cannot read beyond this point.

When sequencer writers update their `watermark`, they also check that other sequencer watermarks are updated promptly. If a sequencer node has not updated its watermark within a configurable interval, it is marked as offline and the watermark is no longer included in the query for the minimum event timestamp. Future events from the offline sequencer are ignored after this timestamp.

Note: For this process to operate optimally, the clocks of the hosts of the sequencer nodes should be synchronized. This is considered reasonable for co-located sequencer hosts which use NTP.

Recovering Sequencer Nodes

When a failed sequencer recovers and resumes operation, it deletes all events that arrived past its last known watermark. This avoids incorrectly re-inserting them, as readers may have seen them already.

It is safe to do this and it does not affect events that have already been read. Any events written by the sequencer while it is offline are ignored by readers. The sequencer then replaces its old watermark with a new timestamp and resumes normal operation.

After resuming operation, there is a short pause in reading from other sequencers due to updates to the watermark table. However, requests to the other sequencer nodes continue successfully, and any events written during this period are available for reading as soon as the pause is over.

The recovered sequencer has likely lost any send requests that were in process during failure. These can be safely retried, without the risk of creating duplicate events, once their `max-sequencing-time` is exceeded.

Mediator

Like the *participant node*, the mediator is replicated and only one replica node is active.

All replicas of the same mediator node share the same database to access the state and coordinate with the active mediator node.

1.25.2 HA for Production Systems

This section covers how to implement High Availability (HA) in production systems.

1.25.2.1 HA and Horizontal Scaling

Introduction

This section describes how to deploy a complete Daml solution in an HA configuration with horizontal scaling characteristics.

The distributed solution uses Daml v2.x Enterprise with Canton services, HTTP JSON API server, Trigger services, and OAuth 2.0 middleware components. In this document we primarily discuss a SQL domain that uses PostgreSQL as the synchronization mechanism for the sequencer backend. We also describe a blockchain domain.

Information in this section is useful for the following:

Production deployment planning.
Understanding HA architectures.
Understanding Daml application scalability.
Building Kubernetes deployments.
HA/scalability deployments in the cloud or on-premises.

Target audience

1. A distributed application provider who runs a domain and their own participant service.
2. Distributed application users who run a participant service.
3. Infrastructure operators or site reliability engineers (SREs).

Important: A distributed application build engineer persona acts as a combination of all three target audiences as they need to validate that the distributed application works as part of the CI/CD activity.

1.25.2.2 High Availability From a Business Perspective

Important: This section contains information for those unfamiliar with HA and how it is fundamental to operational efficiency. We look at how business goals drive the configuration and operational aspects of the HA deployment.

Those familiar with these principles may skip this page.

Definition[1]

> High availability (HA) is a characteristic of a system which aims to ensure an agreed level of operational performance, usually uptime, for a higher than normal period.

> There are three principles of systems design in reliability engineering which can help achieve high availability.

[1] https://en.wikipedia.org/wiki/High_availability as retrieved 02/22/2023

1. Elimination of single points of failure. This means adding or building redundancy into the system so that failure of a component does not mean failure of the entire system.
2. Reliable crossover. In redundant systems, the crossover point itself tends to become a single point of failure. Reliable systems must provide for reliable crossover.
3. Detection of failures as they occur. If the two principles above are observed, then a user may never see a failure - but the maintenance activity must.

Daml solution design honors these principles by:

1. Eliminating single points of failure through redundant components.
2. Executing reliable crossover through networking best practices, in conjunction with the Canton transaction consensus protocol, to eliminate partially processed requests.
3. Ensuring automated failover when a single failure is detected.

Useful External Resources

Multi-Region fundamental 1: Understanding the requirements.
Availability Table.
Embracing Risk.
What is an error budget—and why does it matter?.
Available or not? That is the question—CRE life lessons.

Availability

Availability defines whether a system is able to fulfill its intended function over a period of time, i.e. the system works as intended 99.5% or 99.999% of the time.

The inverse is the percentage of time it is expected to fail, such as 0.5% or 0.001%.

Time-based availability

Availability is usually measured in whole system uptime percentage, rather than the uptime percentages of separate components.

A refinement of this metric is *unplanned downtime*, i.e. the amount of time that the system is unexpectedly unavailable. This is because well-published maintenance activities have no business impact whereas unplanned downtime can cause lost revenue, reputational harm, customers switching to a competitor, etc.

The general formula is:

$$availability = uptime/(uptime + downtime)$$

This formula calculates how many minutes of downtime are allowed in a given period. For example, a system with an availability target of 99.99% can be down for up to 52.56 minutes in an entire year and stay within its availability level.

The table below shows estimated downtimes for a number of given availability levels.

Availability level	Downtime per year	Downtime per quarter	Downtime per month	Downtime per week	Downtime per day | Downtime per hour	
90%	36.52 days	9.13 days	3.04 days	16.80 hours	2.40 hours | 6.00 minutes	
95%	18.26 days	4.57 days	1.52 days	8.40 hours	1.20 hours	3.00 minutes
99%	3.65 days	21.91 hours	7.30 hours	1.68 hours	14.40 minutes	36.00 seconds
99.5%	1.83 days	10.96 hours	3.65 hours	50.40 minutes	7.20 minutes	18.00 seconds
99.9%	8.77 hours	2.19 hours	43.83 minutes	10.08 minutes	1.44 minutes	3.60 seconds
99.95%	4.38 hours	1.10 hours	21.91 minutes	5.04 minutes	43.20 seconds	1.80 seconds
99.99%	52.59 minutes	13.15 minutes	4.38 minutes	1.01 minutes	8.64 seconds	0.36 seconds
99.999%	5.26 minutes	1.31 minutes	26.30 seconds	6.05 seconds	0.86 seconds	0.04 seconds

Note: For a custom availability percentage, use the availability calculator.

Data like this helps a business define an error budget or the maximum amount of time that a technical system can fail without contractual consequences. [2] which may also be a KPI for SREs.

For example, over a 30 day (43,200 minutes) time-window, with an availability target of 99.9%, the system must not be down for more than 43.2 minutes. This 43.2 minute figure is a concrete target to plan around, and is often referred to as the error budget. If you exceed 43.2 minutes of downtime over 30 days, you fail to meet your availability goal.

Aggregate request availability

In contrast to time-based availability, the fine-grained aggregate request availability metric considers the number of failed requests i.e. x% of total failed requests.

This metric is most useful for services that may be partially available or whose load varies over the course of a day or week rather than remaining constant, or to monitor specific, business-critical endpoints.

The general formula is:

$$availability = successfulRequests/totalRequests$$

Although not all requests have equal business value, this metric is often calculated over all requests made to the system. For example, a system that serves 2.5M requests per day, with a daily availability target of 99.99%, can serve up to 250 errors and still hit the target.

[2] https://www.atlassian.com/incident-management/kpis/error-budget

Note: If a failing request retries and succeeds, it is not considered failed since the end-user sees no failure.

Resiliency

Resiliency is related to availability. Resiliency is the capability to handle partial failures while continuing to execute and not crash. In modern application architectures — whether it be microservices running in containers on-premises or applications running in the cloud — failures are going to occur. For example, applications that communicate over networks (like services talking to a database or an API) are subject to transient failures. These temporary faults cause lesser amounts of downtime due to timeouts, overloaded resources, networking hiccups, and other problems that come and go and are hard to reproduce. These failures are usually self-correcting. [3]

Resiliency and availability are enhanced by best practice patterns, such as the retry pattern. When a customer submits a request and receives a success response, they expect that request to succeed. If they receive an error response instead, then the user does not expect it to succeed and knows that they need to retry the request.

Retries can be an effective way to handle transient failures that occur with cross-component communication in a system. [?] A retry pattern is often coupled with the circuit breaker pattern, which effectively shuts down all retries on an operation after a set number of retries have failed. This allows the system to recover from failed retries after hitting a known limit and gives it a chance to react in another way, like falling back to a cached value or returning a message to the user to try again later. [?]

The Daml solution's client application needs to add this type of resiliency to increase availability of the overall system consisting of platform and application.

Other Common Metrics / RTO and RPO

Recovery Time Objective (RTO) is the maximum acceptable delay between the interruption of service and restoration of service. This value determines an acceptable duration over which the service is impaired. It is a slice of the error budget but for a single instance of downtime.

Recovery Point Objective (RPO) is the maximum acceptable amount of time since the last data recovery point. This determines the acceptable data loss between the latest recovery point and a service interruption.

Financial systems often require support for an RPO of zero.

[3] https://azure.microsoft.com/en-us/blog/using-the-retry-pattern-to-make-your-cloud-application-more-resilient/

HA Cost Trade-Offs

High availability can be costly and thus require trade-offs.

To illustrate, extreme events that are highly improbable and costly to guard against - such as an asteroid strike that wipes out a continent's data centers - may not need consideration. This highlights the trade-off between the cost of avoiding an outage, the probability of a single failure (single component redundancy), and the probability of multiple simultaneous failures (multiple component, integrated redundancy).

We can analyze the trade-offs by deriving the cost of loss of availability using unplanned downtime as follows:

$$cost = errorBudget * revenueLostPerMinuteOfDowntime$$

where the revenue lost per minute of downtime is a projected or measured statistic.

Use this formula in different configurations to compare increasing cost against availability to determine an appropriate trade-off for your business goals.

1.25.2.3 Use Cases By Role

Distributed Application Provider

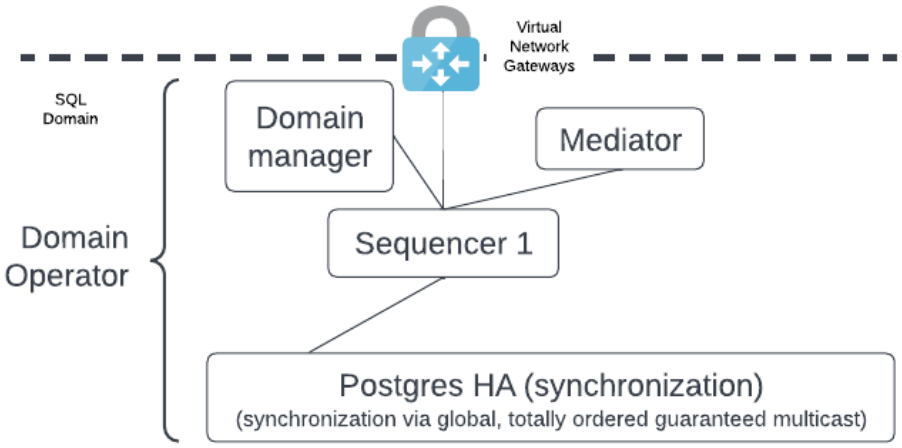
The distributed application provider is also the domain owner and the domain administrator. Their deployment activities come first since all other activities require a domain.

Deploy a domain

The distributed application provider deploys the following components:

The domain manager.
The mediator.
The sequencer.
The HA-configured PostgreSQL managed service[1] that is the sequencer's backend.

[1] The PostgreSQL managed service could also be a PostgreSQL server running on hardware that was deployed by the user.

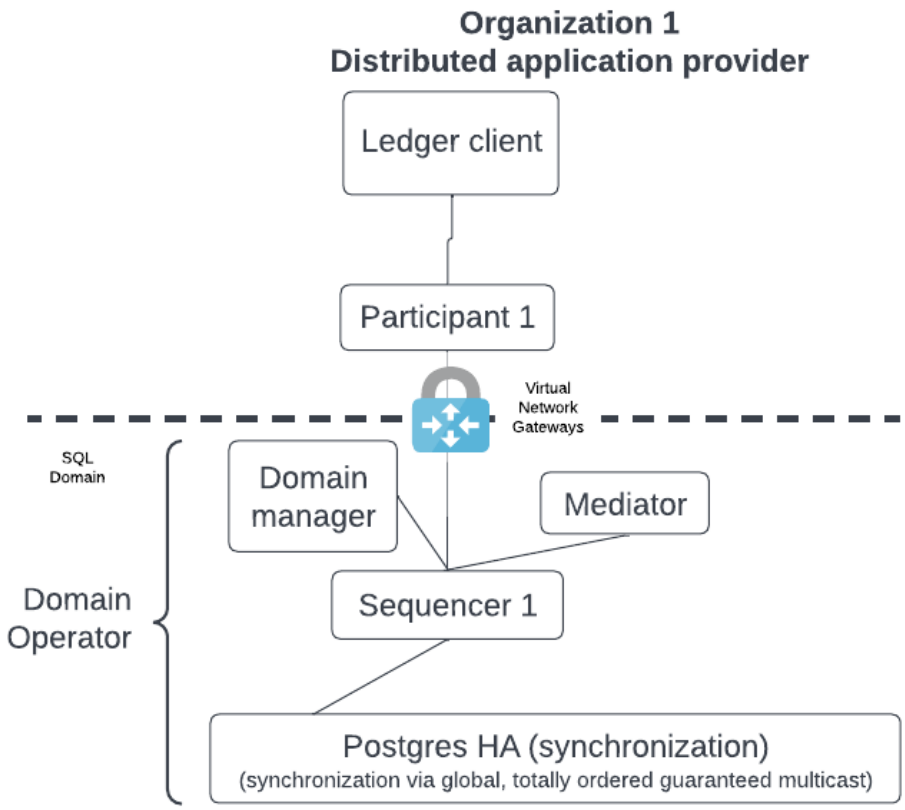


Note:

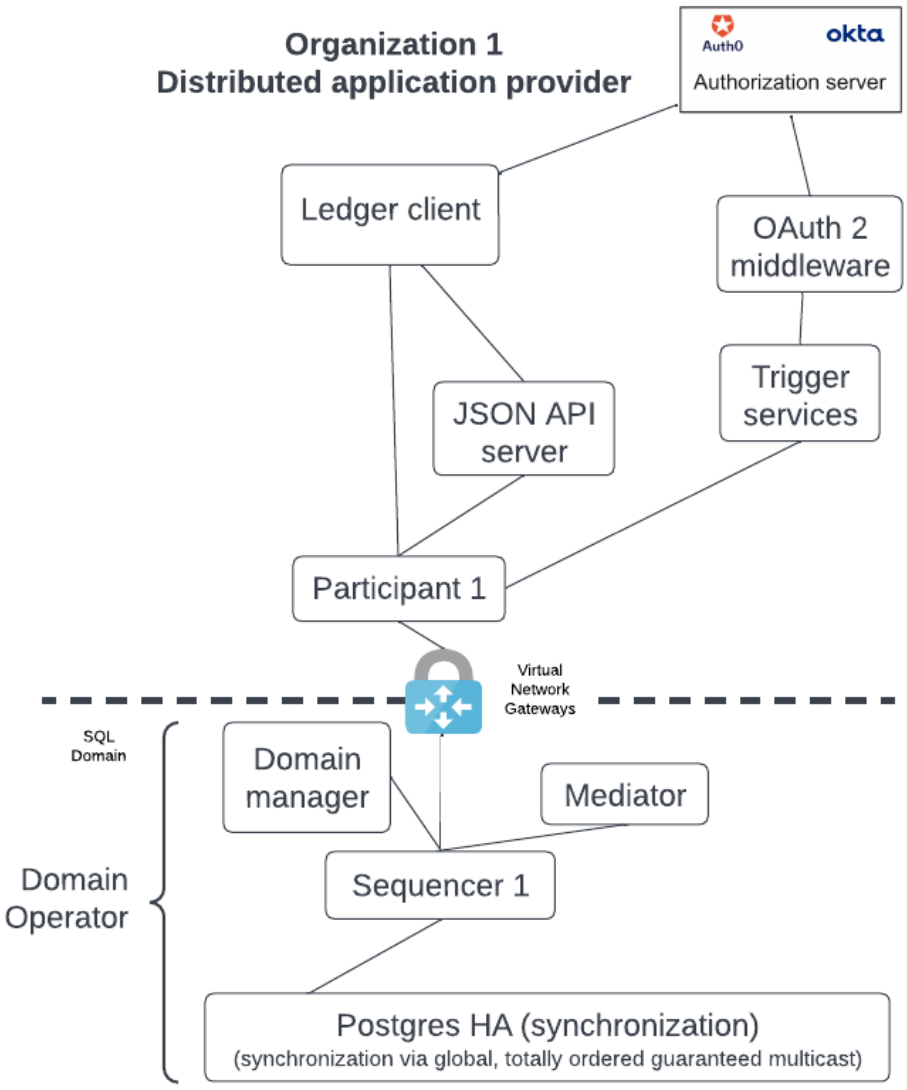
The domain manager, mediator, and sequencer all have internal databases - not shown here - which should be HA-configured.
Also not shown, a bastion host (e.g. Azure bastion host) can be configured for accessing the domain components. This provides an additional layer of security by limiting access to the domain. Additional production access controls may be needed.

The distributed application provider may choose to isolate the domain from their participant node as a security measure using a Virtual Network Gateway as shown. If this additional isolation is not required then the Virtual Network Gateway is not needed. A different type of networking component may be more appropriate - e.g. HAProxy, NGINX, etc.

The figure below shows the participant node and its ledger client.



As mentioned, the distributed domain owner can add additional components which interact with the participant node. These components are normally deployed shortly after deploying the participant node.

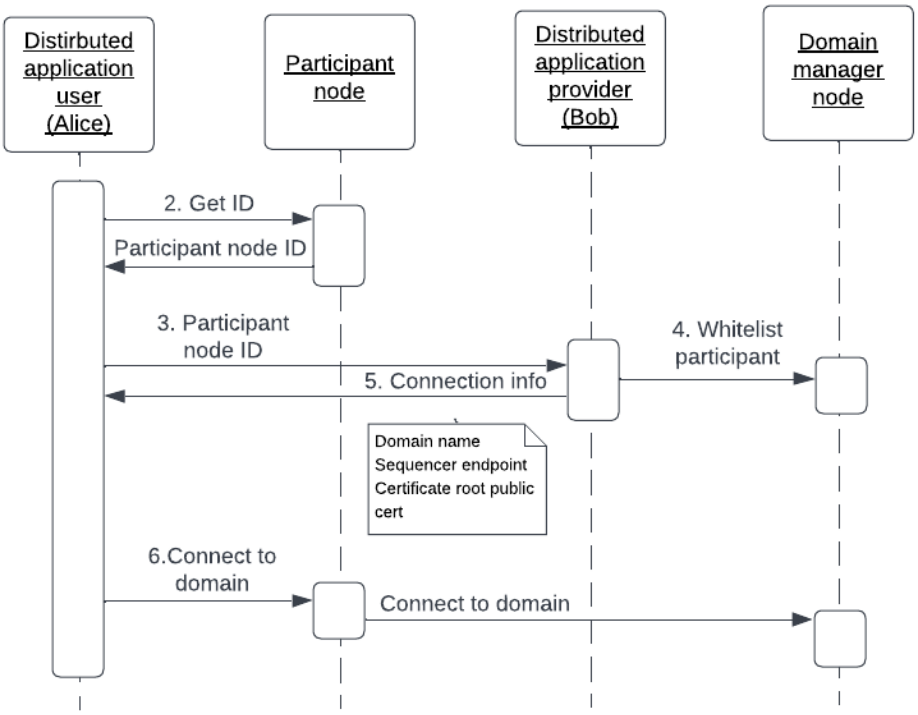


Connect a new participant node

We expect the domain to run in permissioned mode with allow-listing[2] enabled to only include participant nodes whose identities have been registered with the domain manager. This involves a data exchange between the distributed application provider and the distributed application user.

The distributed application provider communicates specific information to a new distributed application user so that the user's participant node can join the application's domain. The figure below illustrates this exchange, with **Bob** as the application provider and **Alice** as the new application user.

[2] The default mode is an open mode which is less secure.



1. Alice deploys a participant node - not shown.
2. Alice extracts the participant node's unique identifier into a string. The id includes the display name for the participant plus a hash of the public identity signing key.
3. Alice makes her participant id known to Bob through an external mechanism, e.g. email.
4. Bob runs a console command which adds Alice's participant id to the domain allowlist and configures the appropriate node's permissions. An example command which gives default permissions is shown here:

```
domainManager1.participants.set_state(participantIdFromString,
↪ParticipantPermission.Submission, TrustLevel.Ordinary)
```

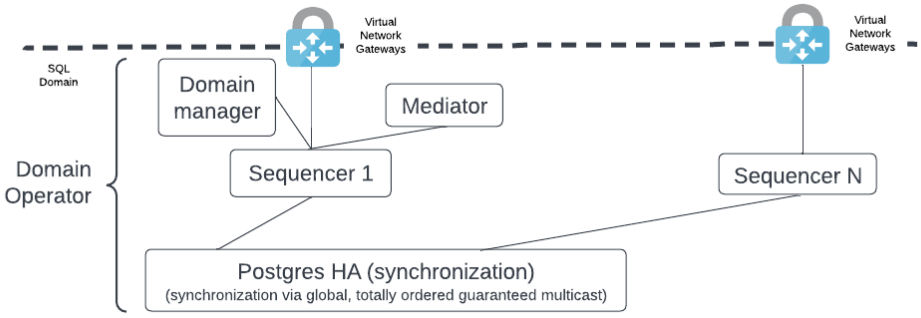
5. **Bob passes Alice the following information, which allows her to connect to the domain:**
   a. One, or more, sequencer endpoints - https urls.
   b. Certificate root public cert, if it's not a publicly signed CA.
6. Alice picks a unique name for the domain that is local to her participant. This will be used in the connection command.
7. Alice enters the information into the connection command `connect_multi` and connects to Bob's domain - not shown.

```
participantAlise.domains.connect_multi("AliceDomainName", Seq(sequencer1,
↪sequencer2))
```

Prepare domain infrastructure for adding new participant nodes

A distributed application provider expands the use of their application by allowing more participant nodes to join their domain. A sequencer node is the gateway to the domain for all participant nodes. It follows that the policy on when to add a new sequencer is important and must be clearly defined.

As shown below, a domain may start with a sequencer node and then add more sequencer nodes as required.



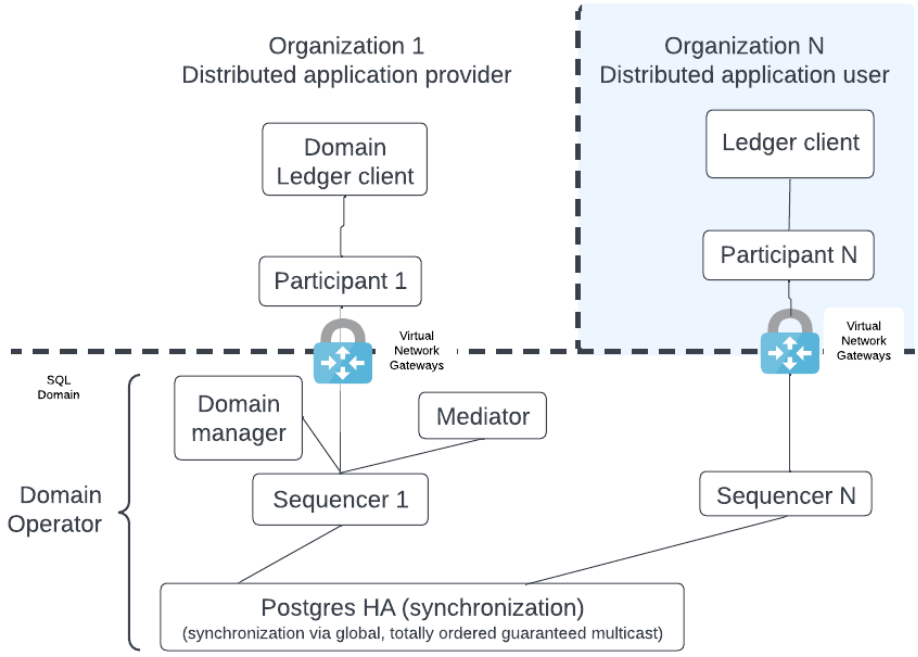
There are two options for adding a new participant node.

1. Deploy a sequencer for each participant node to introduce more isolation between the events each participant sees. For example, Coke may want to avoid cross-contamination of events with Pepsi, and vice-versa, so each organization wants its own sequencer. The Coke sequencer sees all the Coke and Pepsi messages through the shared database; the sequencer backend is a broadcast. However, Coke's sequencer node provides a multicast to Coke's participant node with only Coke's events. Pepsi's setup functions similarly.
2. Avoid the additional isolation and focus on high resource utilization of the sequencer by having several participant nodes use the same sequencer; i.e. a single sequencer handles multiple distributed application users. This option produces a lighter load on the joint HA PostgreSQL database.

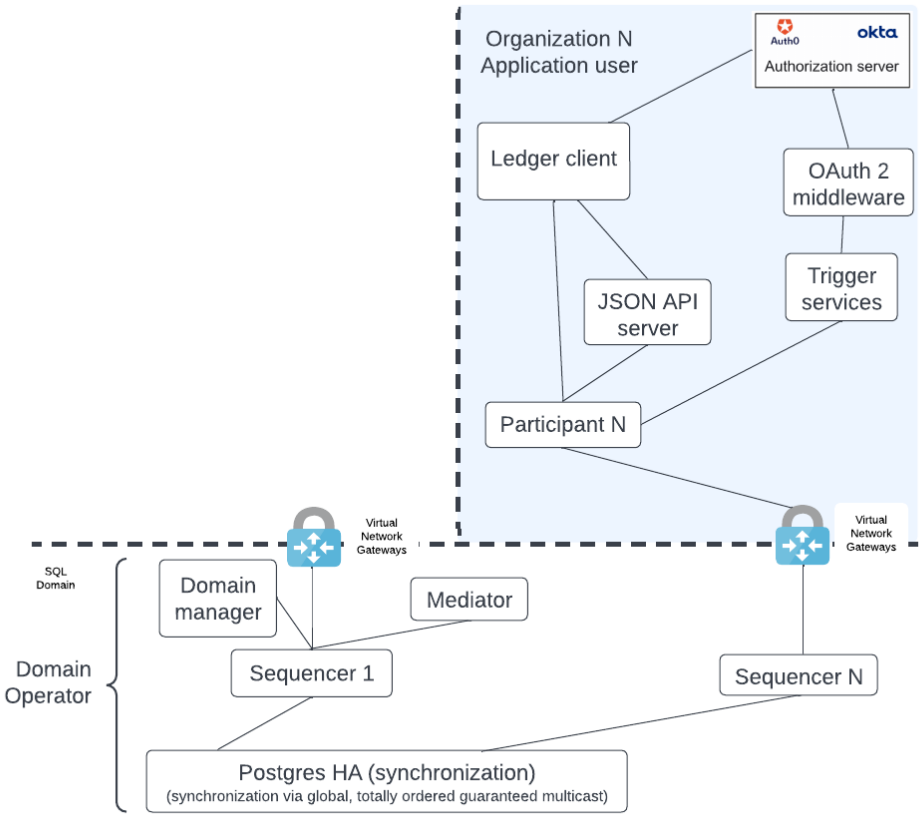
Distributed Application User

The distributed application user deploys their own participant node and connects to the provider's public sequencer endpoint. There is some similarity here with the distributed application provider. However, the distributed application user's DAR files (i.e. business logic) may be a subset of the DAR files deployed by the distributed application provider.

This setup is extendable. For example, the distributed application user may be interested in several distributed applications, and so connect their participant node to the related domains by deploying multiple DARs for the different applications' business logic. They may also write their own extensions that include additional DARs. These extensions do not impact the use cases described here.



The simple configuration shown above, like that of the domain owner, can expand into a more capable deployment such as shown below by adding the HTTP JSON API server, trigger services, and OAuth2 middleware.



Upload the distributed application DAR files

Check the documentation for information on how to *upload DAR files*.

Site Reliability Engineer (SRE)

Monitor systems

The SRE's primary use case is monitoring. Monitoring is required on both the domain and participant nodes, although the scope is slightly different.

Monitoring normally consists of the following activities:

Export logs.
Expose metrics via Prometheus endpoint.
Parse out trace IDs from the log files.
Keep logs for audit.

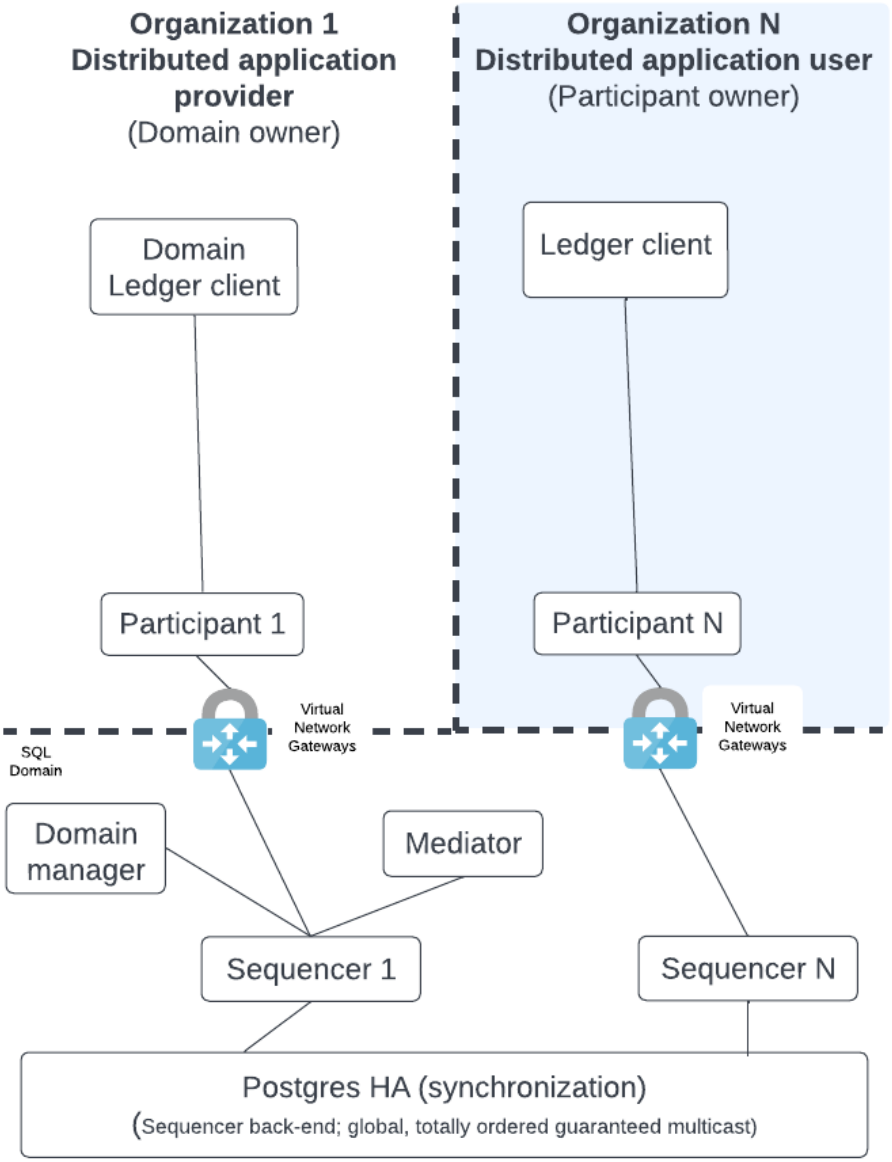
Check the documentation for more information on *monitoring*.

1.25.2.4 Implementing HA and Scaling Deployments

Basic Daml Deployment

The diagram below illustrates the most basic multi-party Daml deployment possible.

Each logical box in the diagram contains multiple internal components in an HA production configuration. The following sections expand on each of these logical boxes to show how they are configured for production.



The diagram shows the following components:

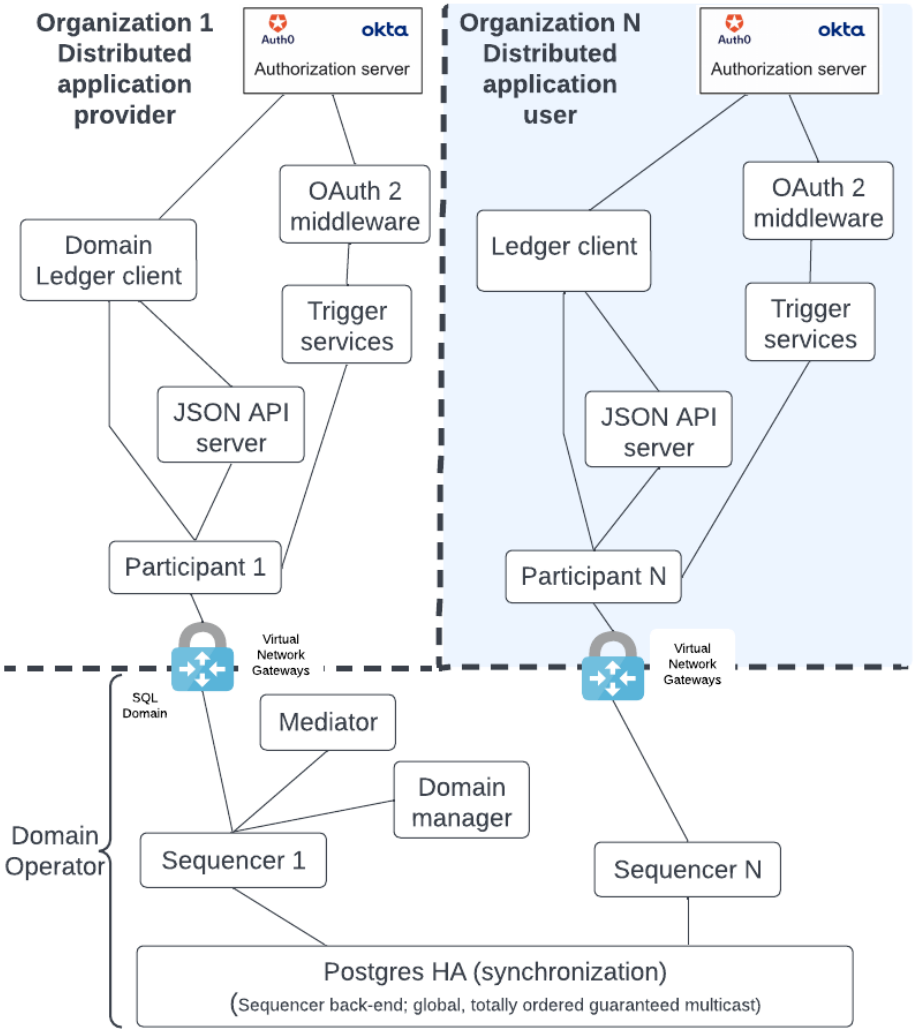
A **Ledger client** that uses the Ledger API; the client entry point to execute business logic.
Participant nodes which expose the public Ledger API. They execute the Daml business logic of the distributed application based on an API request or as part of the Canton transaction consensus protocol.
A **Mediator** which acts as a transaction manager for the Canton consensus protocol. Ensures that either all of the parts of a transaction succeed or there is no change.
The **Domain manager** which manages the domain with transactions that update the topology and make public keys available.
Sequencers expose the Canton API so that all clients see events as ordered by a guaranteed, multicast communication mechanism. A sequencer has a backend component that is hidden from its clients. Depending on the backend component, the solution supports either a SQL or blockchain domain.

Note: The term **node** may refer to a logical box with multiple components or a single JVM process depending on context.

The distributed application **provider** deploys several components: the domain (domain manager[1], mediator, and sequencer) and their own participant node(s).

The distributed application **user** only has to deploy a participant node and connect that node (from their own private network) to the private network of the domain via communication with a sequencer.[2]

A typical Daml deployment has additional components which are shown in the figure below:



The diagram shows the following components:

[1] The domain manager can also be referred to as the 'topology manager'. For a production deployment, the domain manager can be thought of as containing the topology manager with some additional capabilities.

[2] Although there are multiple sequencers shown, this is just for illustration purpose. As little as a single sequencer is needed. For example, Organization N's participant node could connect to Sequencer 1 and not Sequencer N.

An HTTP **JSON API server** which supplements the gRPC API endpoints of the participant node by providing an HTTP REST (HTTP JSON API) endpoint. It also has an internal cache so that it can be more responsive to queries.
Trigger services that listen to the ledger event stream for events that trigger business logic.
OAuth2 middleware that supports a refresh of the Trigger services JWT token and manages the background requests for a refresh token for the Trigger services.
The *Identity Provider (IDP)* is the authentication entity that provides the JWT token.. The IDP is outside of the Daml solution but nevertheless a necessary component. Different organizations may use different IDPs for their participant nodes.

Note: We expect the domain owner to implement additional business logic for managing the distributed application in both their participant node and trigger service nodes.

Architecture for HA and Scaling

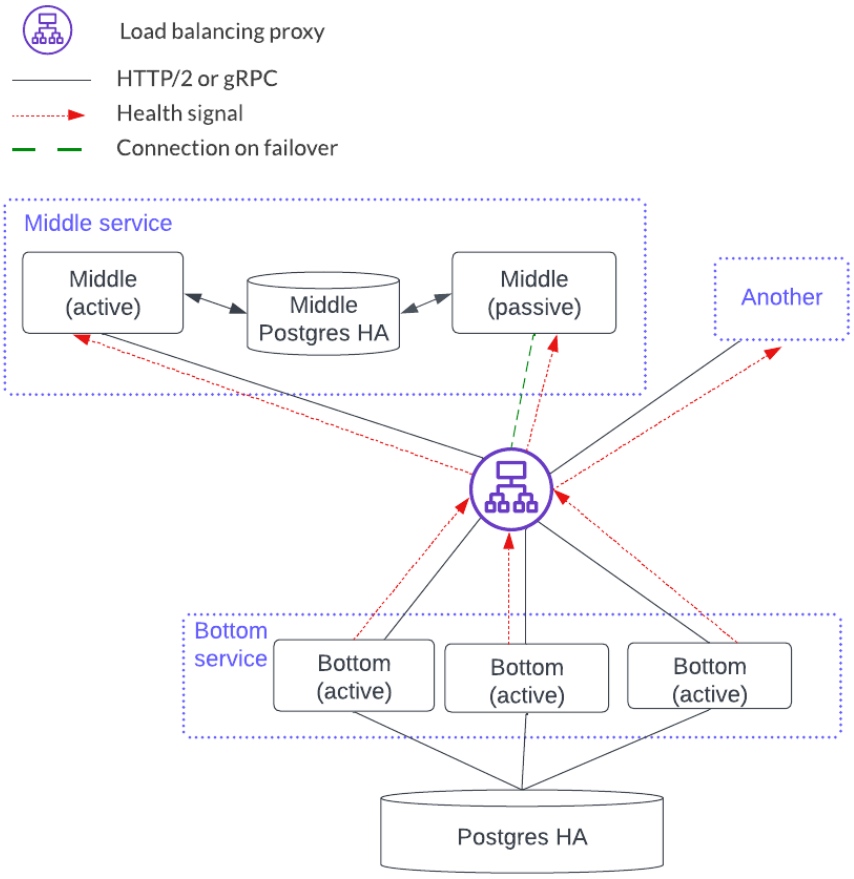
As a production system becomes busier, it is necessary to scale up the components.

Vertical scaling is the easiest way to handle more load, but there are limits to its benefits. Vertical scaling is not discussed here since this is a well-known technique. Instead, this document focuses on horizontal scaling where backup/redundant components are deployed to different availability zones as part of the HA configuration.

Note: For clarity the diagrams follow these conventions:

Solid, black boxes for individual instances, processes, and containers.
Databases may be identified as shared and highly available with an **HA** in the disk figure.
Distinguishing between a single instance and the HA variant is done by using the term **service** for HA. There is also a blue dashed line around the components that make up an HA service. The word **service** is chosen because it looks like a single endpoint which is highly available, like a managed service in the cloud.
For simplicity, a blue dashed box with a name is shorthand for the HA variant of that component.
Health signals are a dashed red line that point to the instance that is a recipient of that signal.
Communication channels that are passive but become active upon failover are bordered by a dashed green line.

Thus, in the figure below, the **Middle service** blue box encompasses all the components that make up that service. Middle services instances are in black boxes with solid lines. The blue box **Another** is short form for a service called Another. There is a load balancer between the middle and bottom services.



Each component can scale using a stateless or stateful horizontal scaling pattern. In this diagram, the bottom service has instances that are independent and considered stateless. Stateless horizontal scaling is achieved by adding another bottom instance. This also increases the availability because there are more redundant instances. The middle service is stateful since the instances share a local database so the HA model is active-passive. Scaling the stateful middle service is achieved by replicating the entire middle service: i.e. add two instances connected by a PostgreSQL HA database.

HTTP JSON API and Participant Node Services

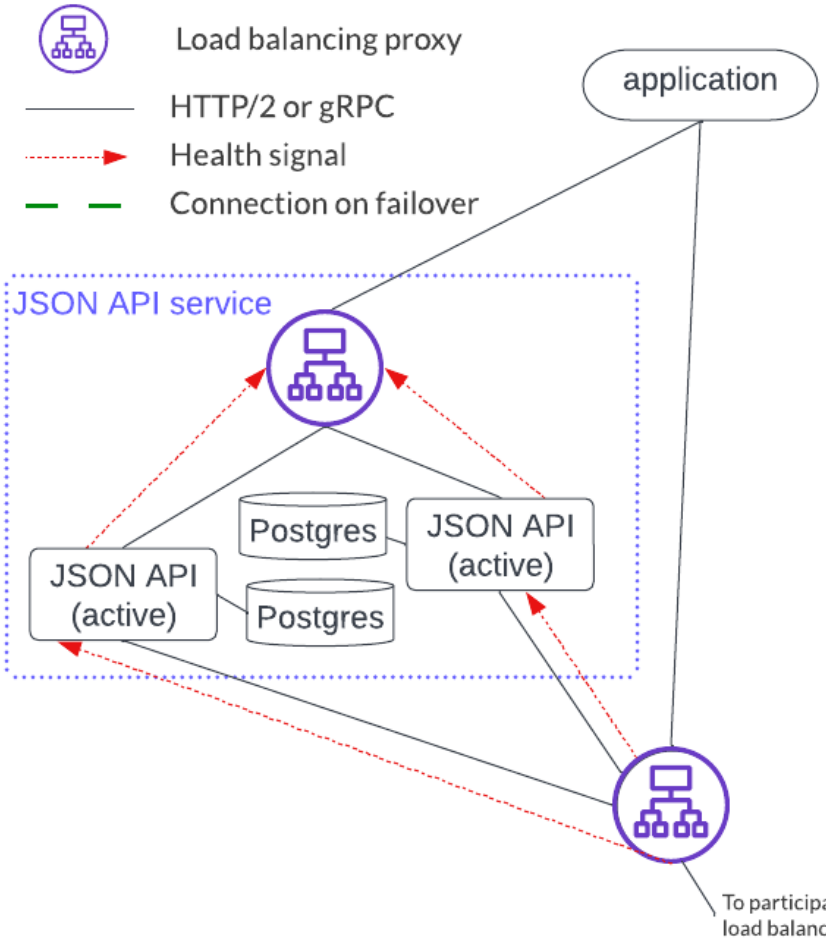
In cases where there is a single participant service (and corresponding HTTP JSON API service) that all the client requests go to, the HTTP JSON API and participant services need to be considered together since there are some state dependencies between the two. In particular, users and related parties are configured on a participant node so they will be handled by a particular participant service. This means that the HTTP JSON API service that is connected to a participant service also serves those same users and parties.

However, if there is more than one participant service (e.g. with horizontal scaling) then it is the application's responsibility to understand which participant service to send a request to (and the corresponding HTTP JSON API service), based on the user(s) or parties of the request. Another way to describe this is that users and parties are sharded across the participant and HTTP JSON API services and the application is responsible for targeting the right instance.

As shown below, an HTTP JSON API service is an endpoint that has four components. Each HTTP JSON

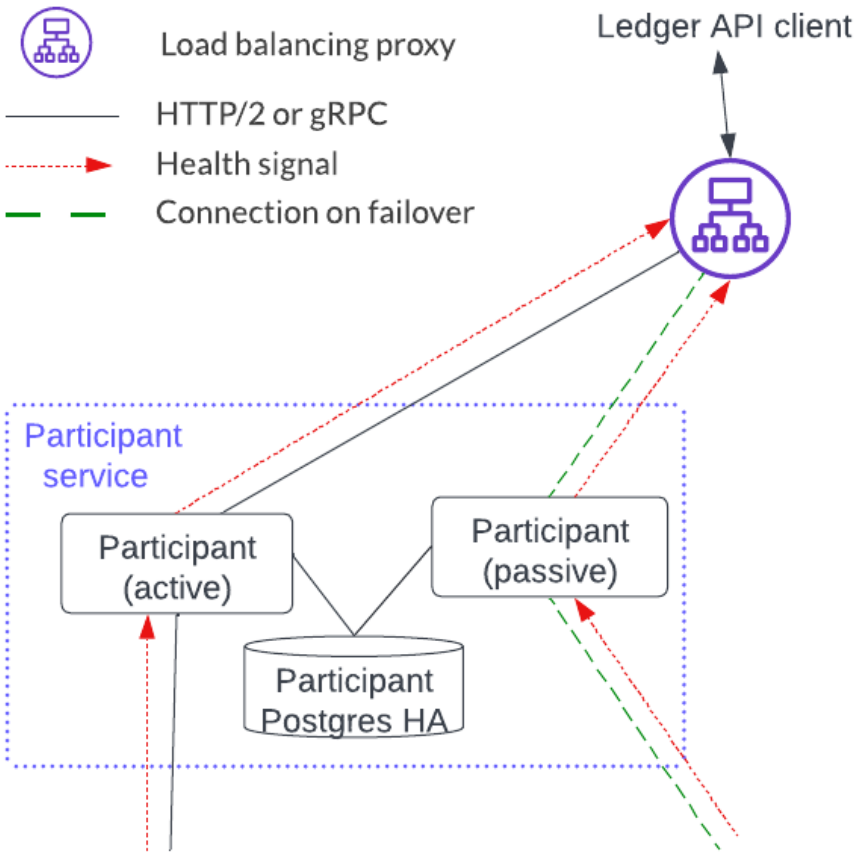
API instance emits a health signal that the load balancer uses to direct traffic. The HTTP JSON API's database acts as a cache that is local to the instance, meaning it does not need to be HA since the cache can be reconstructed at any time.

Note: The HTTP JSON API server does not currently support mTLS from client applications. mTLS is supported between the load balancer and participant node.



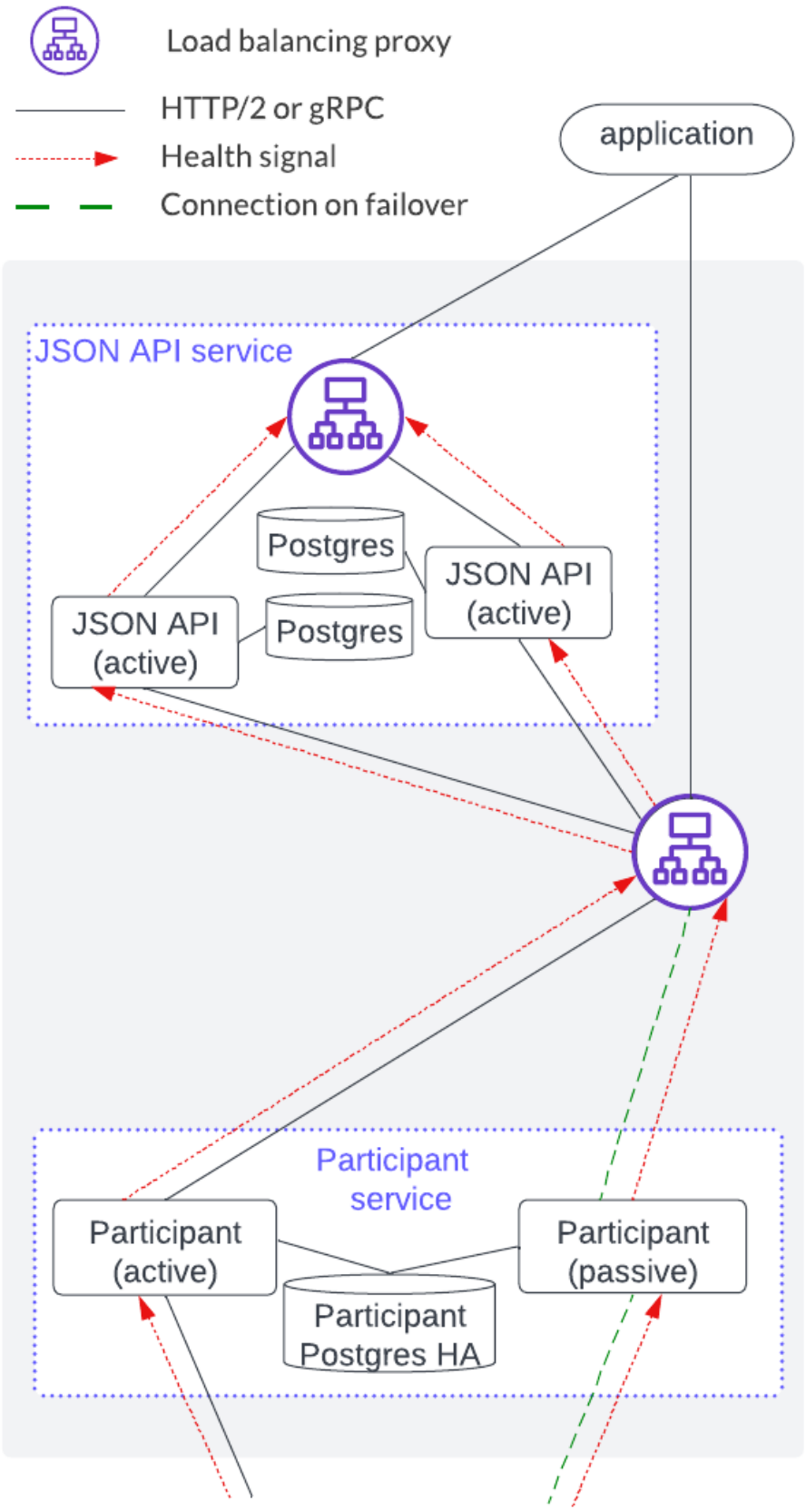
There are a couple of important distinctions between the participant service and the HTTP JSON API service setup:

A single participant service can have several HTTP JSON API servers. However, a given HTTP JSON API server should only connect to a single participant service.
The HTTP JSON API component operates in an active-active mode while participant nodes operate in an active-passive mode.

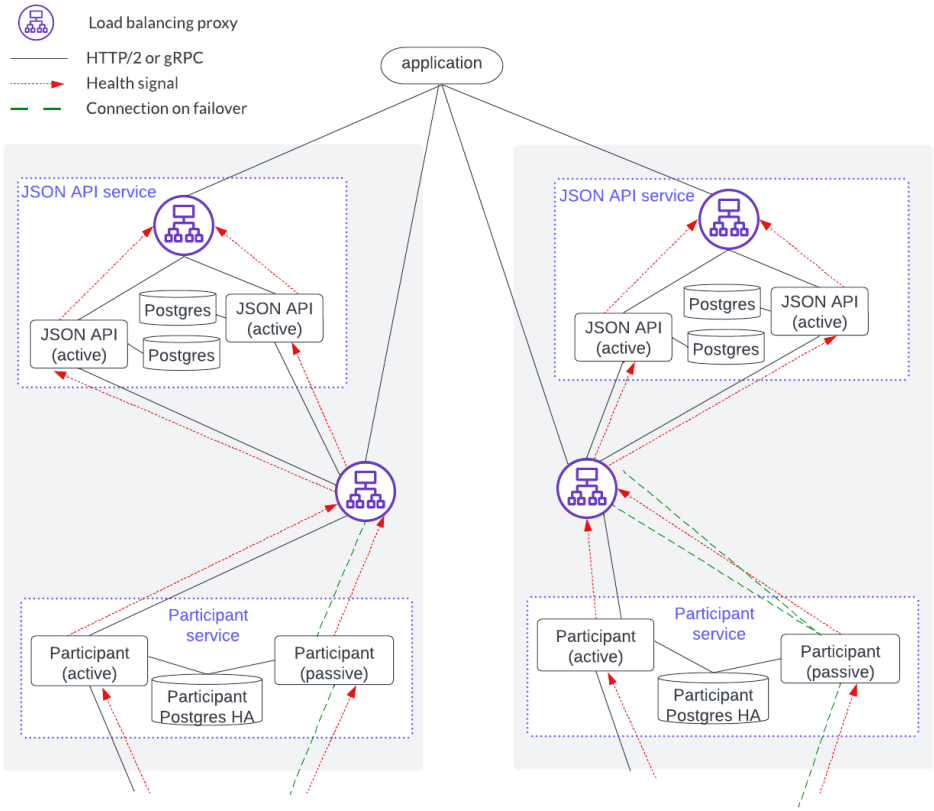


The deployment below shows a single HTTP JSON API service and participant service. There are some hidden state dependencies that include:

A ledger offset that requires the HTTP JSON API server be associated with a single participant service.
Command deduplication functions on a single participant service alone.
Shared users and parties for both the HTTP JSON API service and the participant service.


Load balancing proxy
HTTP/2 or gRPC
Health signal
Connection on failover
application
JSON API service
Postgres
JSON API
(active)
JSON API
(active)
Postgres
Participant
service
Participant
(active)
Participant
(passive)
Participant
Postgres HA

Horizontal scaling is accomplished by sharding application users and parties across a joint HTTP JSON API and participant service, and adding another HTTP JSON API and participant stack, as shown below.



Sequencer Service

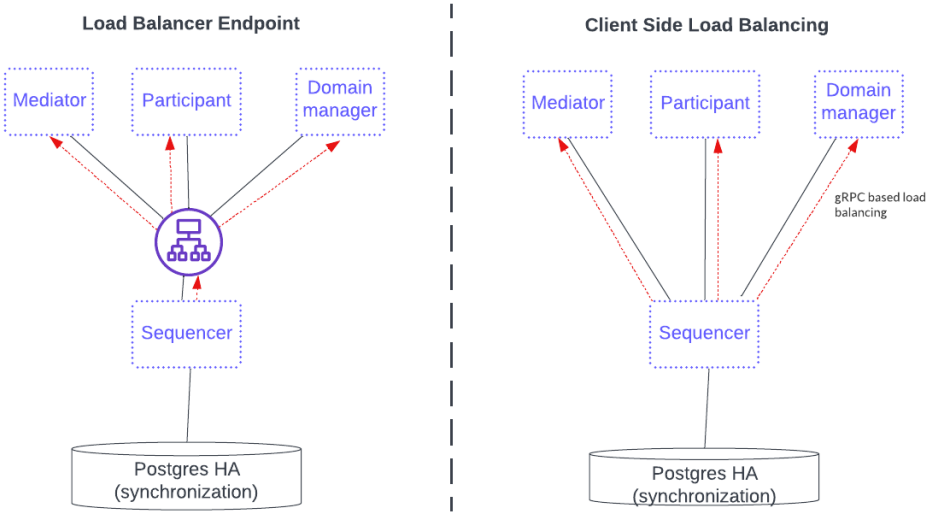
The sequencer service operates in active-active mode, which means that all sequencer instances can accept and process Canton protocol API requests. This has benefits for both scaling and availability.

Deploying a sequencer depends on business requirements which may impact deployment configurations such as load balancing configurations and whether the domain is fully or only partially decentralized.

Sequencer service load balancing options

The sequencer service has several clients: participant, mediator, and domain manager. mTLS between these clients is unavailable at the time of writing.

The two available load balancing options are shown in the diagram below.



The first option, on the left, fronts the sequencer service with a load balancer that all sequencer clients use. This option simplifies configuration and connectivity but adds the complexity of configuring the load balancer.

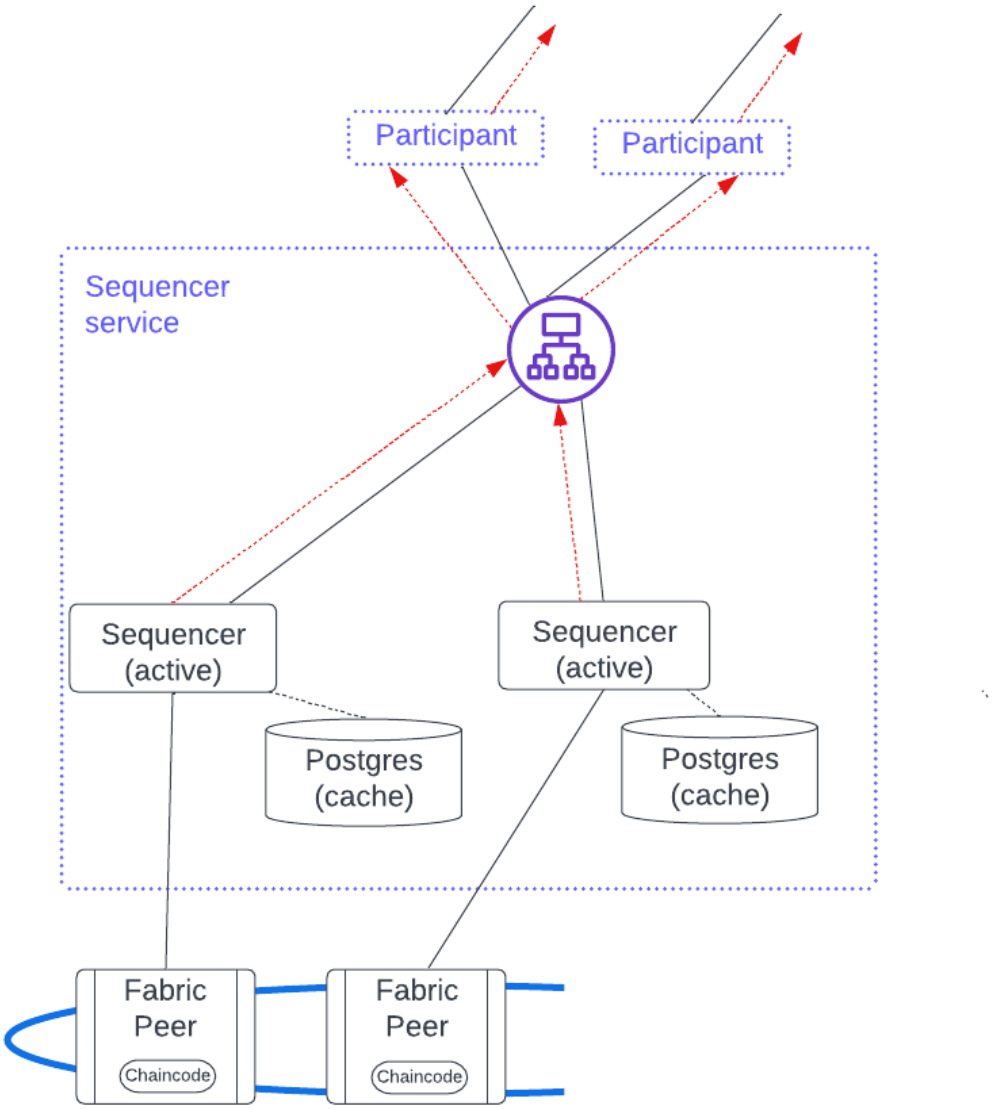
The option on the right is a gRPC java client library providing a round-robin selection mechanism for load balancing that automatically round-robins through multiple sequencer connections and includes the ones that are healthy. This setup requires the distributed application provider and distributed application users to maintain the configuration information of all the available sequencers in the sequencer client. The sequencer client continuously monitors the health of each sequencer endpoint when selecting a possible node in round-robin fashion.

See the Canton documentation on *connection to high availability sequencers* and client load balancing for more information.

Blockchain domains

A blockchain domain has a fully decentralized data path and is used when there is no trust between the distributed application providers and users. Whereas the sequencer queries the PostgreSQL backend directly in a SQL domain, this cannot be done in a blockchain domain. Instead, a local database to the sequencer is added to speed things up. The sequencer backend then uses the blockchain to provide a guaranteed ordered multicast of events.

The figure below shows a HyperLedger Fabric blockchain example. Notice that each sequencer has an independent local cache running on a PostgreSQL database. This local cache ensures efficiency because the sequencer avoids having to scan the entire blockchain when it starts up or reconnects after a temporary interruption. It also reduces the performance load on the blockchain.



This figure has a load balancer fronting the sequencer nodes, but client side load balancing would also work. There are several benefits to using a load balancer:

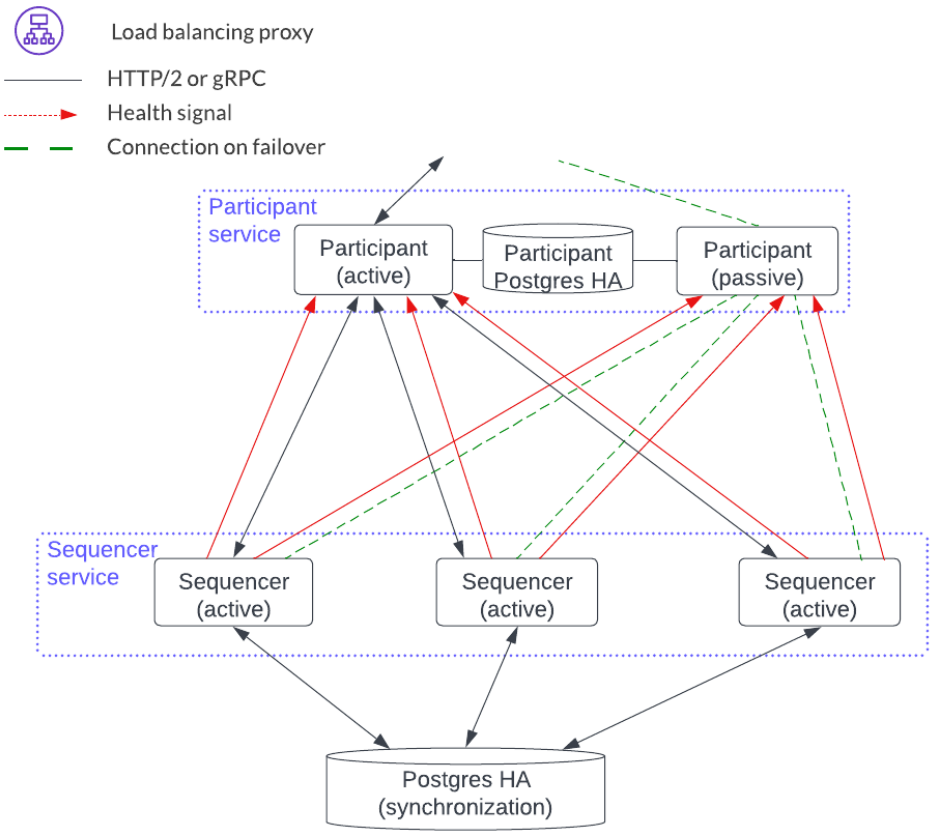
Clients have a single endpoint that consolidates the health signals, simplifying setup and troubleshooting.
Adding a sequencer does not require updating the configuration information in each client.
Additional security.

Since sequencer nodes are always active, horizontal scaling for a blockchain sequencer service is achieved by adding a new sequencer along with its associated local cache database and enabling it for client use.

SQL domains

The SQL domain is only partially decentralized and is used when the sequencer's backend data is stored in a single PostgreSQL database managed by a centralized distributed application provider. This option requires participant users to have some trust in the application provider.

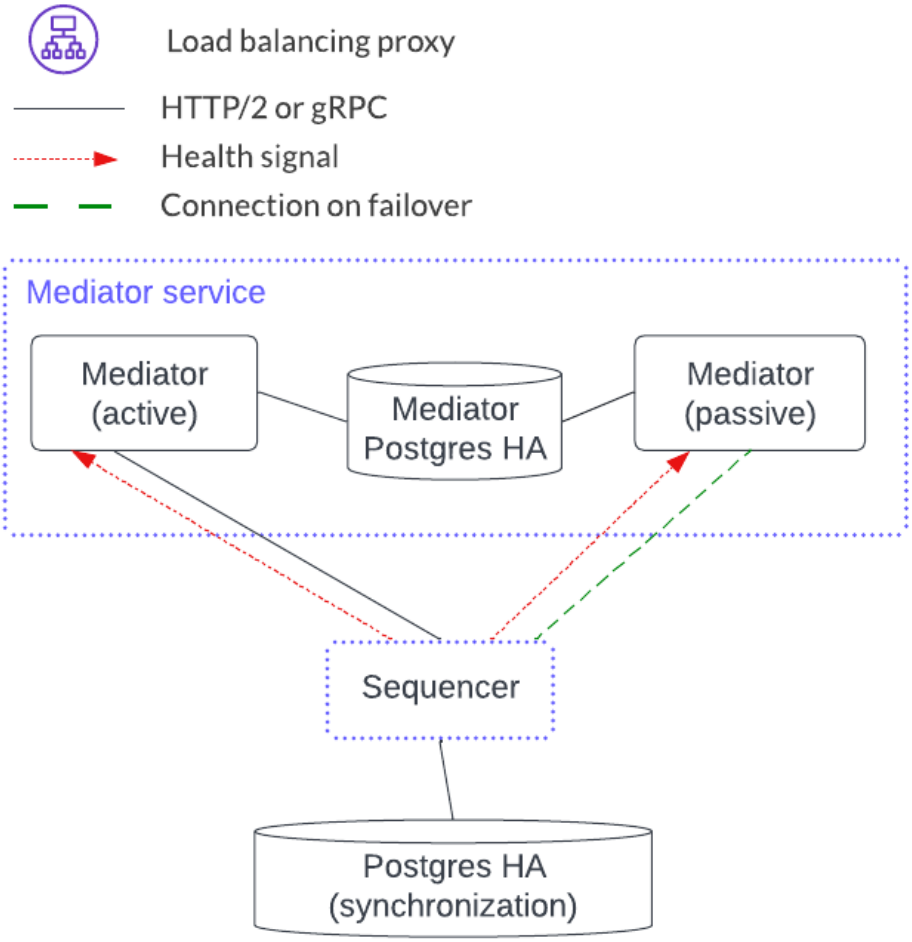
A sequencer needs no local cache because it queries the backend database directly with no performance penalty.



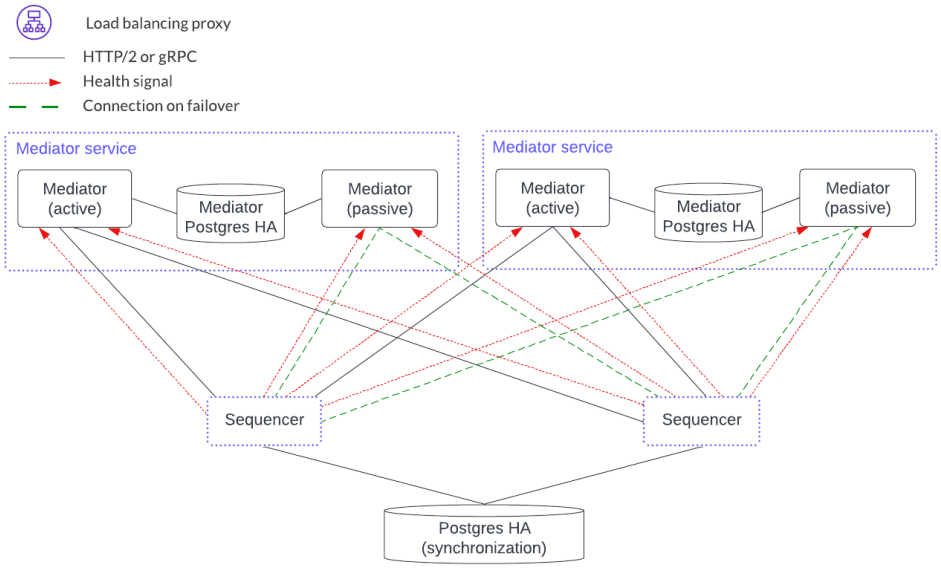
Since sequencer nodes are always active, horizontal scaling for the SQL domain sequencer service is achieved by adding a new sequencer and enabling the clients to use it.

Mediator Service

The mediator service has no client-facing ingest. It also has no load balancing proxy or health endpoints. Instead, it uses client side load balancing based on the gRPC infrastructure. It is like the participant node in that it has a PostgreSQL database in an HA configuration. The mediator components, however, act in an active-passive configuration.

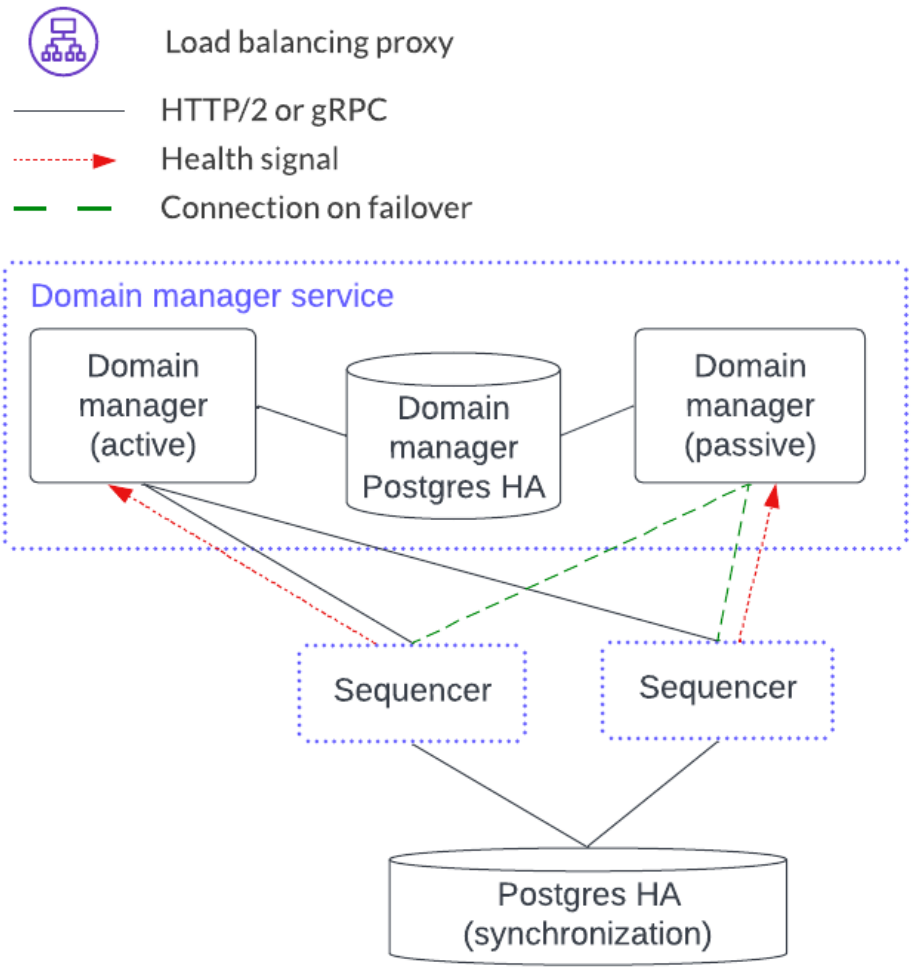


Horizontal scaling is achieved by adding another mediator service.



Domain Manager Service

The domain manager service also has no client-facing ingest point. Like the mediator services, the domain manager is in an active-passive configuration. There is, however, only a single domain manager service per domain. This means that there is no horizontal load balancing model for the domain manager. This is feasible because the domain manager is not in the transaction processing path path and so it manages topology transactions which are orders of magnitude less frequent than the Daml transactions that the mediators manage.



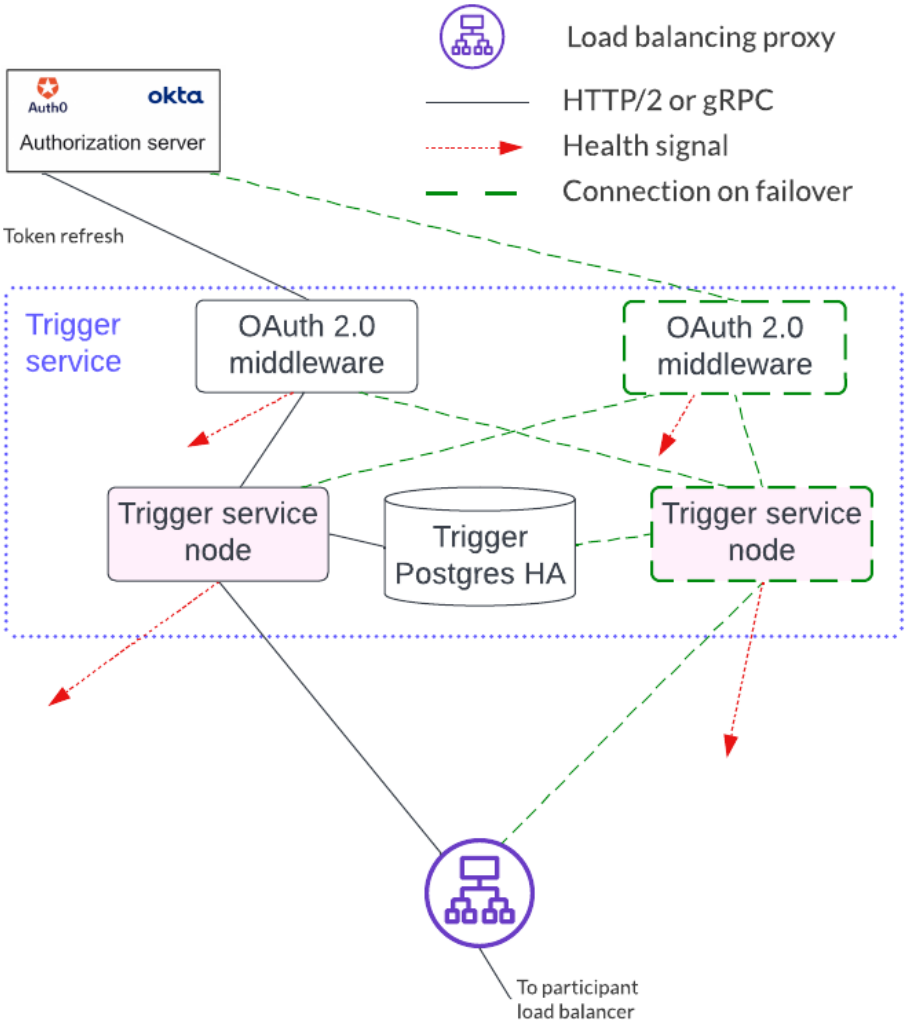
As of v2.5.0, the domain manager uses PostgreSQL in an HA configuration for HA support.

Trigger Service

The trigger service includes the OAuth 2.0 middleware and trigger service nodes. As shown below, it does not operate in an HA configuration that supports a single failure. Instead, it requires a monitoring system to detect if the trigger service node or OAuth 2.0 middleware is unhealthy and mitigate any issues by doing one of the following:

1. Restarting the failed item.
2. Stopping the unhealthy instance and then starting another instance.

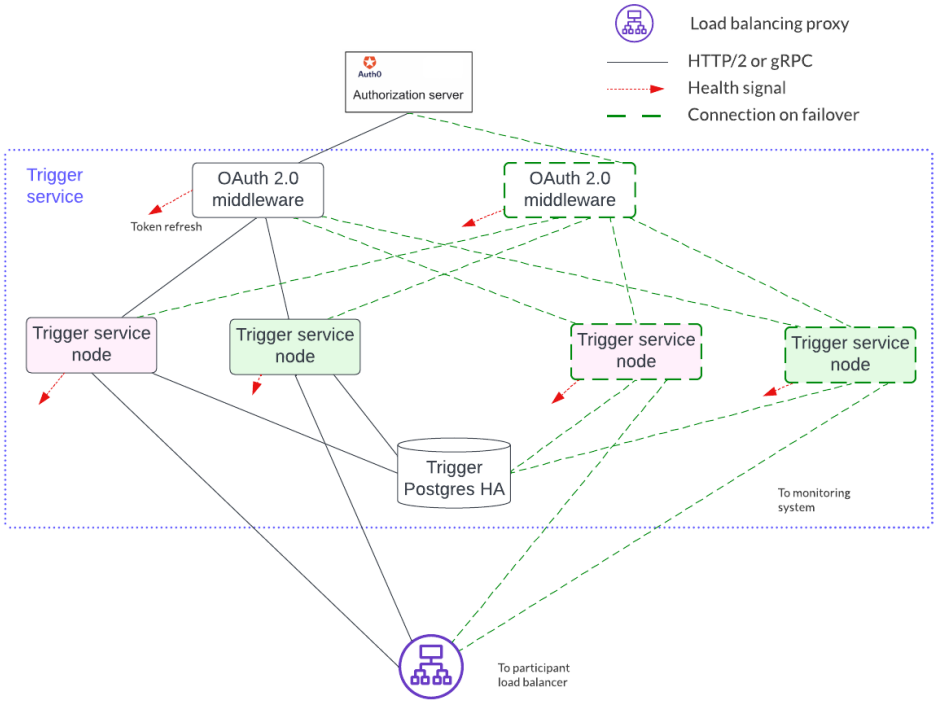
A shared PostgreSQL database is needed for the trigger service node. As shown below, the OAuth 2.0 middleware connects to an OAuth provider.



Horizontal scaling is achieved by deploying additional trigger service nodes. For example, in the figure below, there are two pairs of trigger service nodes (pink and green) which use the same OAuth 2.0 middleware node that is connected to a single OAuth provider.

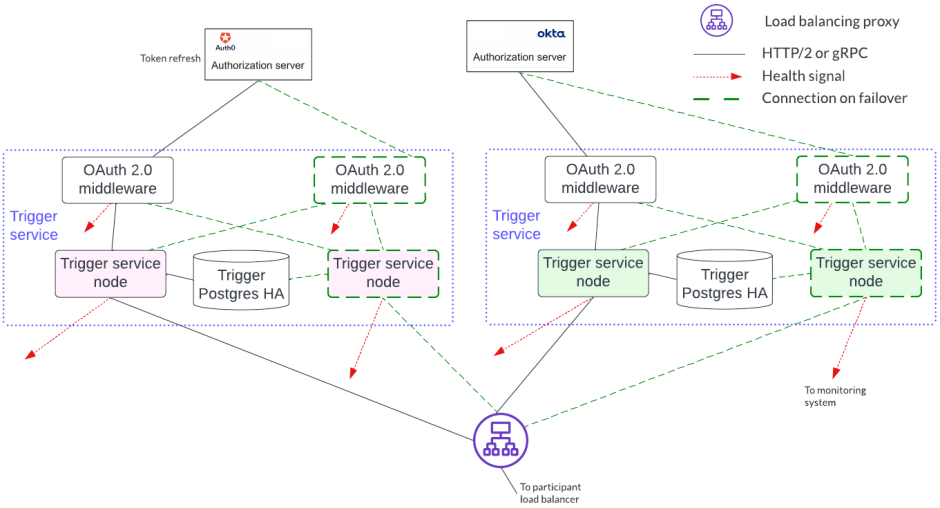
Running the same trigger rule on multiple live trigger service instances is not allowed. In this example the pink rules are running in a single live trigger service node, just like the green rules are running in a single live trigger service node.

Remember, the box with the dashed lines indicate that the node is started when the active node is identified as unhealthy.



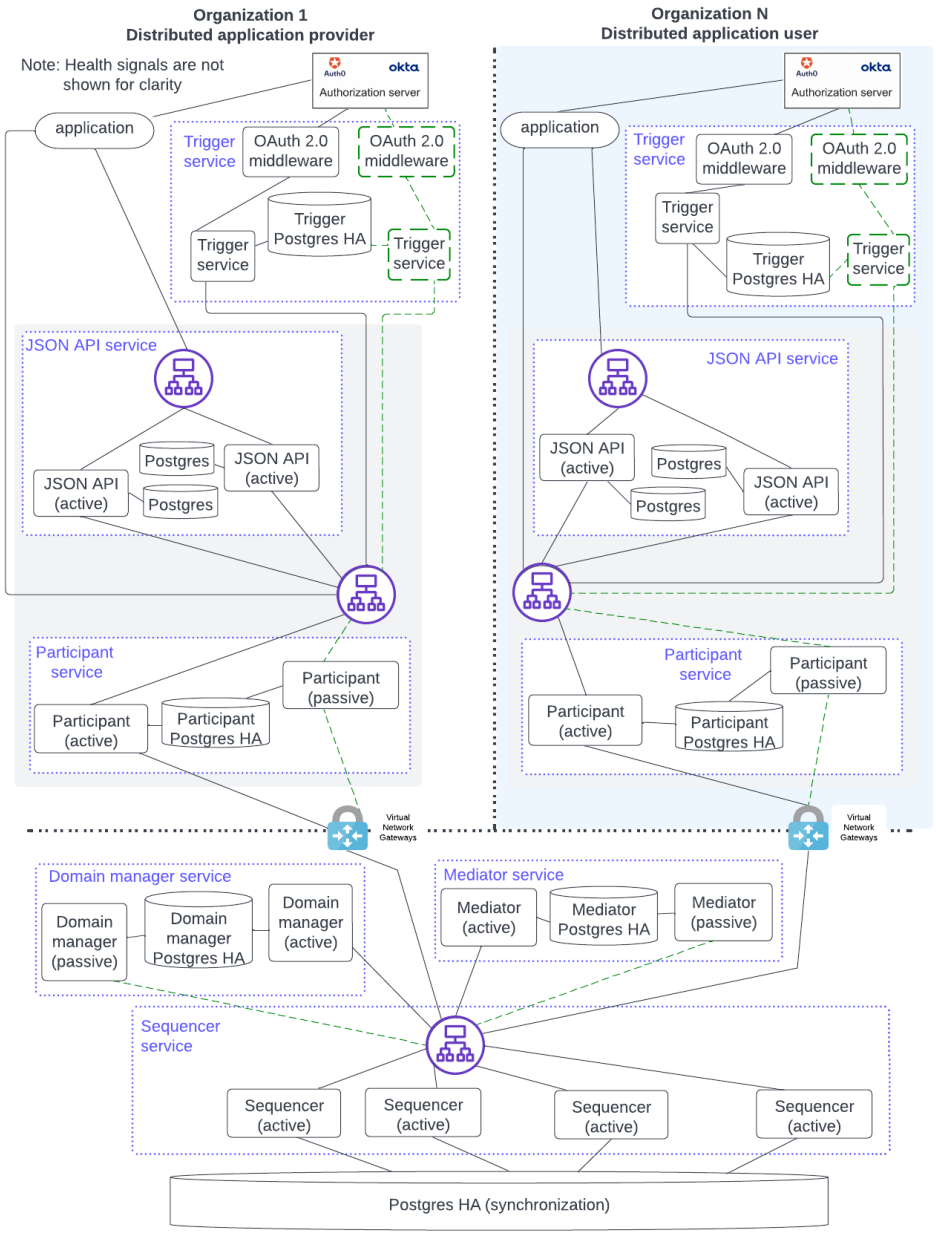
Each trigger service node is limited to a single OAuth provider and is unable to support queries against multiple OAuth providers. For example, the pink and green trigger services in the figure above cannot query against both a Google OAuth provider and an Apple OAuth provider - each trigger service must be configured to use exactly one of these providers.

If access to more than a single OAuth provider is needed, distinct pairs of trigger service nodes and OAuth 2.0 middleware servers are configured. This is shown below. Please note running the same trigger rule on multiple live trigger service instances is not allowed in this configuration either.

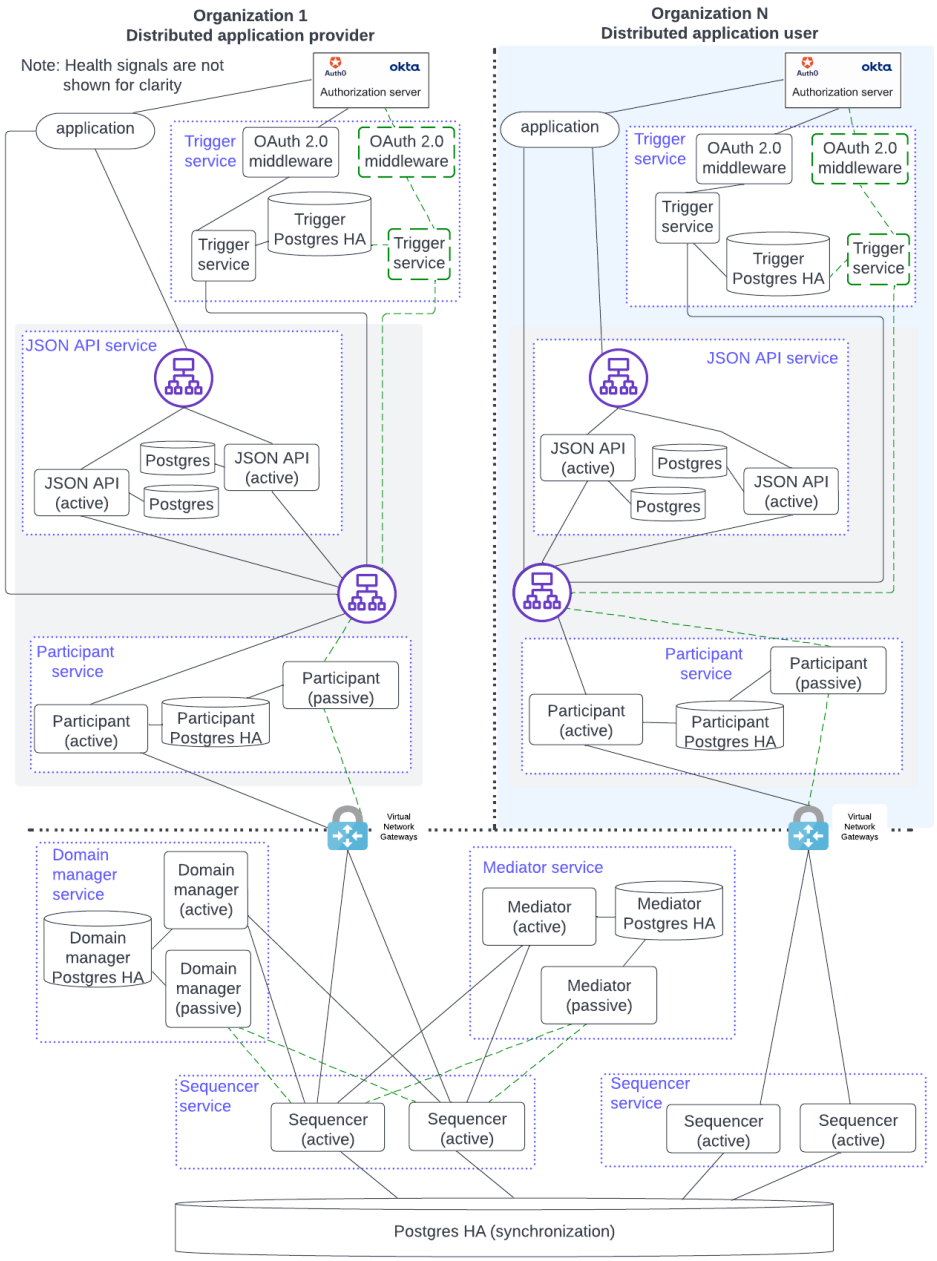


1.25.2.5 HA Deployment Solution for Production

The figure below assembles the components already described using the single-endpoint load balancer option. Although this setup may look complex, each service is independent and deployed separately.



The figure below uses client-side load balancing for the domain owner's sequencer access. Separate sequencer nodes are shown for the distributed application user's connectivity.



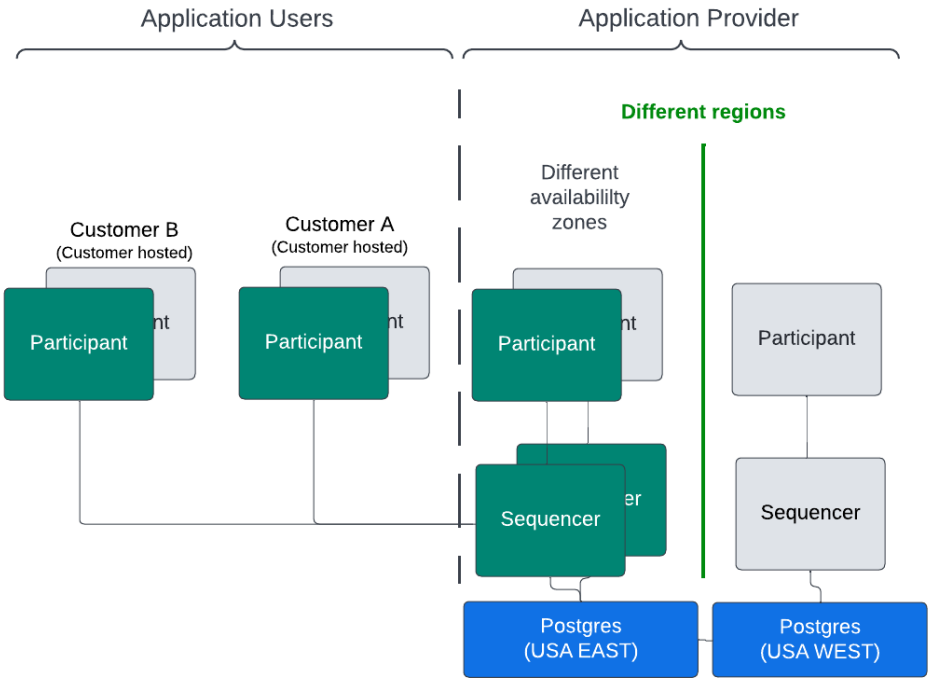
The diagrams maximize the independence between components by showing them as running on independent hosts. However, for actual deployment scenarios, some simplification and cost reduction is possible. For example, combining components on the same host is a decision that reduces complexity and cost but may impact availability if one component impacts another (e.g. when one component uses 100% of the CPU and starves the other components).

Distinct service instances should, in principle, run on different hosts to avoid a single point of failure at the infrastructure level. However, business goals always drive the HA requirements and how things are deployed.

1.25.2.6 HA in the Cloud

The HA deployment and horizontal scaling models already discussed are generic by design and focus on handling single component failures automatically and transparently. A cloud deployment, along with orchestration tools, adds additional HA capabilities and options for more complex failure modes.

The figure below shows a minimal, high-level, AWS-based HA solution. The active nodes are green and the passive nodes are gray. Different availability zones can house different instances of the components within a service to provide location resiliency. For example, if the active participant node in US-EAST-1 fails then the passive node in US-EAST-2 becomes the active node.



Note: Network connectivity between the relevant components is not shown.

Having redundant components in different regions creates additional location resiliency. For example, in the figure above, an active participant node is deployed in the USA EAST region and a passive participant node in the USA WEST region. The redundant, passive participant may not even be running depending on how the HA solution has been architected to satisfy the business requirements, such as:

> The entire Daml solution stack may switch over to a different region all at once with a global load balancer redirecting the requests to the newly activated region. This can address the situation where a normally active region becomes unavailable.
> Single components may be started in different regions for a finer-grained HA approach. This introduces additional network latency for cross-region traffic.
> Directing a switchover from one region to another is atypical and adds complexity so this may be manually initiated, or require manual approval, to avoid flapping from one region to another when a problem is intermittent.

> The sequencer backend is an HA database that can work across regions. The options are discussed below. Sequencers in an availability zone can be running since they act in an active-active mode.

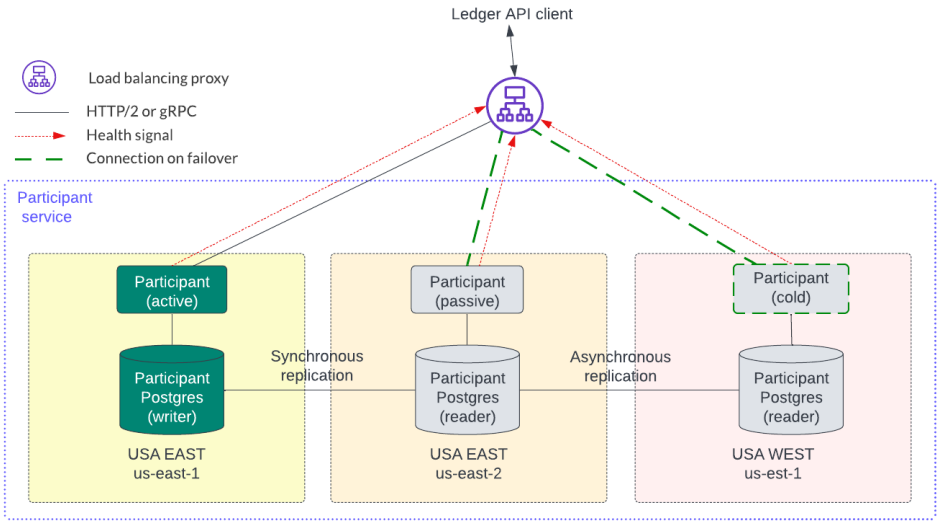
A redundant sequencer in a different region may be cold and need to be started if the PostgreSQL database it is connected to is read-only. The sequencer backend database in the example is PostgreSQL operating in a highly available manner with a single write node and read-only replicas. However, the read-only replicas and write nodes use synchronous replication to avoid data loss - the sequencer backend can look like a ledger fork to participant nodes if there is data loss.

Per AWS:

> When writes involve synchronous replication across multiple Regions to meet strong consistency requirements, write latency increases by an order of magnitude. A higher write latency is not something that can typically be retro-fitted into an application without significant changes.[1]

Synchronous replication impacts latency and throughput so tuning and testing are needed.

Although not shown in the figure above, the databases for each service may need to be highly available and shared across availability zones and regions. To illustrate this, the participant service is expanded into three regions in the figure below. The latencies within a region are expected to be low so synchronous replication within a region provides an RPO of zero for single failures. Asynchronous replication for a participant node can be used across regions but it can incur some data loss as described in the restore caveats documentation.

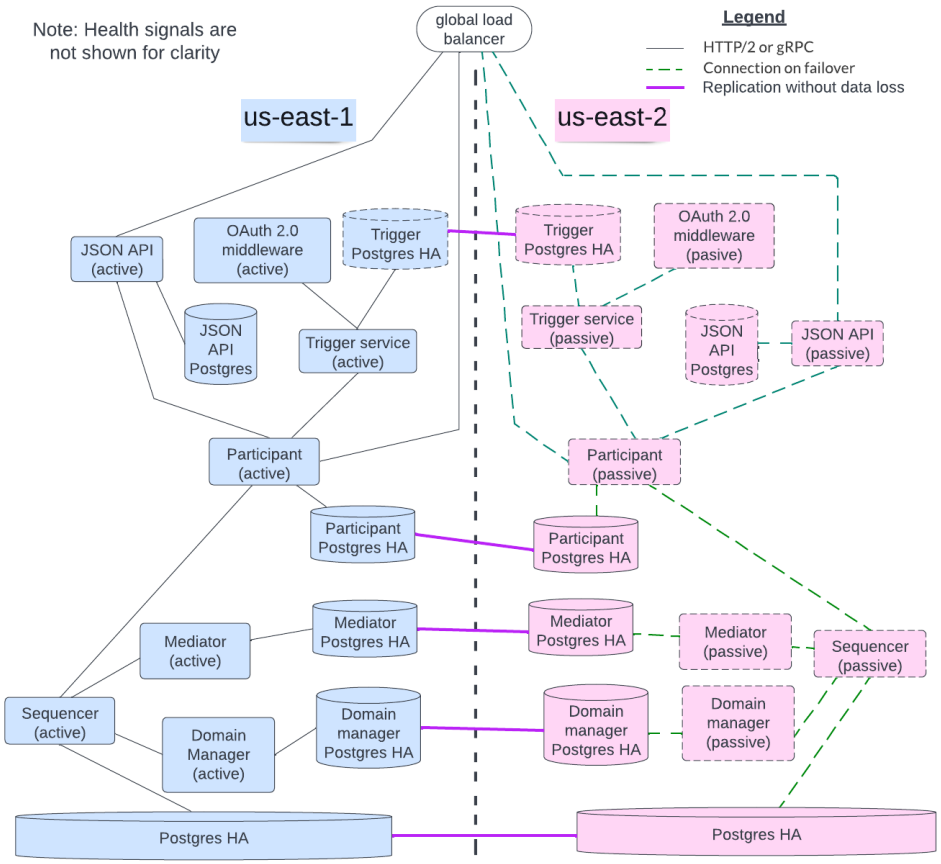


The initial block diagram in this section expands into the larger figure below which shows all the services acting in HA mode. The sequencer backend, participant, mediator, and domain manager nodes all have replicated databases ensuring no data loss.

By leveraging the elasticity of the cloud, the orchestration tool may provide possible cost reduction, at the expense of additional recovery time, by not running the passive node instances in the background. Instead, the orchestration tool starts a passive node when it detects the active node is unhealthy or has failed. In general, the node startup time is typically several seconds. However, additional time may be needed for additional data synchronization. Passive nodes can also be running

[1] https://docs.aws.amazon.com/whitepapers/latest/aws-multi-region-fundamentals/multi-region-fundamental-2-understandin
html

in standby mode but this incurs the cost of running those nodes.



When there is a failover in the Daml solution, some requests may not succeed. Specifically, with the Canton transaction consensus protocol either a request completes in its entirety or there are no changes. This means that, although there is no cleanup required for failed requests, the application is responsible for retrying the failed request that did not complete during a failover event. The application needs to be designed to handle this scenario (which is a common requirement for web-based applications).

See the documentation on the metrics RTO and RPO for more information.

Database Options

Each cloud vendor chooses from several PostgreSQL options. Selection is ultimately driven by business requirements, which drive the HA requirements fulfilled by selecting the appropriate PostgreSQL option. A managed database selection allows for trade-offs in availability if choosing between an Aurora DB cluster or an Aurora global database. Amazon RDS for PostgreSQL is a self-managed option which is more flexible than the managed service. Each of these options is introduced below to explore what each can provide in an HA context.

Although the examples presented here are for AWS, other cloud vendors have similar technologies that are compatible with PostgreSQL. Please consult the relevant cloud vendors documentation.

Amazon RDS for PostgreSQL, Multi-AZ with two readable standbys[Page 1025, 2]

This is a self-managed option for deploying:

> highly available, durable MySQL or PostgreSQL databases in three AZs using Amazon RDS Multi-AZ with two readable standbys. Gain automatic failovers in typically under 35 seconds, up to 2x faster transaction commit latency compared to Amazon RDS Multi-AZ with one standby, additional read capacity, and a choice of AWS Graviton2 - or Intel-based instances for compute.[Page 1025, 2]
>
> Amazon Aurora database provides RPO zero at the storage level by requiring at least four of the six storage nodes to acknowledge receipt before confirming the transaction. Aurora splits the six storage nodes across Availability Zones (AZs) in an AWS Region. Amazon Relational Database Service (Amazon RDS) Multi-AZ (except SQL Server) provides close to RPO zero at the storage level independently of the database. It writes each block synchronously to two Amazon Elastic Block Storage (Amazon EBS) volumes in two different AZs.[3]

Amazon Aurora DB cluster[Page 1025, 4]

This option:

> consists of one or more DB instances and a cluster volume that manages the data for those DB instances. An Aurora cluster volume is a virtual database storage volume that spans multiple Availability Zones, with each Availability Zone having a copy of the DB cluster data.[Page 1025, 4]

Additionally,

> An Aurora Replica is an independent endpoint in an Aurora DB cluster, best used for scaling read operations and increasing availability. An Aurora DB cluster can include up to 15 Aurora Replicas located throughout the Availability Zones of the Aurora DB cluster's AWS Region.[5]

Aurora global database[Page 1025, 6]

This database:

> consists of one primary AWS Region where your data is written, and up to five read-only secondary AWS Regions. You issue write operations directly to the primary DB cluster in the primary AWS Region. Aurora replicates data to the secondary AWS Regions using dedicated infrastructure, with latency typically under a second. An Aurora global database supports two different approaches to failover. [Page 1025, 6]
>
> Recovery from Region-wide outages - The secondary clusters allow you to make an Aurora global database available in a new primary AWS Region more quickly (lower RTO) and with less data loss (lower RPO) than traditional replication solutions.[Page 1025, 6]

[2] https://aws.amazon.com/rds/features/multi-az/
[3] https://aws.amazon.com/blogs/publicsector/a-pragmatic-approach-to-rpo-zero/
[4] https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.Overview.html
[5] https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/AuroraMySQL.Replication.html
[6] https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-global-database.html

Important: This feature is only available in Canton Enterprise

1.25.3 High Availability Usage

This section looks at some of the components already mentioned and supplies useful Canton commands.

1.25.3.1 Domain Manager

As explained in *Domain Architecture and Integrations*, a domain internally comprises a sequencer, a mediator, and a topology manager. When running a simple domain node (configured with `canton.domains`, as shown in most of the examples), this node will be running a topology manager, a sequencer and a mediator all internally.

It is possible however to run sequencer(s) and mediator(s) as standalone nodes, as will be explained in the next topics. But to complete the domain setup, it is also necessary to run a domain manager node (configured with `canton.domain-managers`), which takes care of the bootstrapping of the distributed domain setup and runs the topology manager.

The domain bootstrapping process is explained in *Setting up a Distributed Domain With a Single Console*.

The domain manager can be made highly available by running an active node and an arbitrary number of replicated passive nodes on hot standby, similar to the mediator HA mechanism (see below). The only requirement is shared storage between all the domain manager instances, which must be either Postgres or Oracle. Nodes automatically handle their state and become active/passive whenever the active instance fails, such that from a configuration perspective this is entirely transparent.

An example configuration of a standalone HA domain manager node could therefore simply look like this:

```
canton {
  domain-managers {
    domainManager1 {
      admin-api.port = 5016
      // The storage needs to be either Postgres or Oracle to support replicated
↪domain managers nodes
      // See the persistence section of the documentation for how to set these up
      // https://docs.daml.com/canton/usermanual/persistence.html
      storage = ${_shared.storage}
    }
  }
}
```

In a replicated setup, only the active domain manager can be used to issue topology transactions (for instance bootstrapping a domain or onboard new mediators/sequencers). To find out if a domain manager is active, one can run *domainManager1.health.active* in the canton console (for a domain manager node named *domainManager1*). Another way to avoid this manual check is to place a load balancer in front of the domain managers and let it pick the active instance. See *Load Balancer Configuration* for more information.

Commands that indirectly use the domain manager (for instance connecting a participant to a domain) will automatically be picked up by the active domain manager, so this is only relevant when

issuing commands directly against a specific domain manager.

1.25.3.2 HA Setup on Oracle

The HA approach that is used by the participant, mediator, and sequencer nodes requires additional permissions to be granted on Oracle to the database user.

All replicas of a node must be configured with the same DB user name. The DB user must have the following permissions granted:

```
GRANT EXECUTE ON SYS.DBMS_LOCK TO $username
GRANT SELECT ON V_$LOCK TO $username
GRANT SELECT ON V_$MYSTAT TO $username
```

In the above commands the `$username` must be replaced with the configured DB user name. These permissions allow the DB user to request application-level locks on Oracle, as well as to query the state of locks and its own session information.

For a high-availability deployment the underlying Oracle store must be set up in a highly available manner (for example, using Oracle RAC or Veritas VCS).

Oracle high availability is supported only when the database presents to the Canton nodes as a single, logical Oracle database. There is no support for horizontal scaling through sharding or other multi-database RAC features beyond simple HA clustering.

1.25.3.3 Mediator

The mediator service uses a hot-standby mechanism with an arbitrary number of replicas. During a mediator fail-over, all in-flight requests get purged. As a result, these requests will timeout at the participants. The applications need to retry the underlying commands.

Running a Stand-Alone Mediator Node

A domain may be statically configured with a single embedded mediator node or it may be configured to work with external mediators. Once the domain has been initialized further mediators can be added at runtime.

By default, a domain node will run an embedded mediator node itself. This is useful in simple deployments where all domain functionality can be co-located on a single host. In a distributed setup where domain services are operated over many machines, you can instead configure a domain manager node and bootstrap the domain with mediator(s) running externally.

Mediator nodes can be defined in the same manner as Canton participants and domains.

```
mediators {
  mediator1 {
    admin-api.port = 5017
  }
```

When the domain node starts it will automatically provide the embedded mediator information about the domain. External mediators have to be initialized using runtime administration in order to complete the domain initialization.

HA Configuration

HA mediator support is only available in the Daml Enterprise version of Canton and only PostgreSQL and Oracle-based storage are supported for HA.

Mediator node replicas are configured in the Canton configuration file as individual stand-alone mediator nodes with two required changes for each mediator node replica:

Using the same storage configuration to ensure access to the shared database.
Set `replication.enabled = true` for each mediator node replica.

Note: Starting from canton 2.4.0, mediator replication is enabled by default when using supported storage.

Only the active mediator node replica has to be initialized through the domain bootstrap commands. The passive replicas observe the initialization via the shared database.

Further replicas can be started at runtime without any additional setup. They remain passive until the current active mediator node replica fails.

1.25.3.4 Sequencer

The database-based sequencer can be horizontally scaled and placed behind a load balancer to provide high availability and performance improvements.

Deploy multiple sequencer nodes for the Domain with the following configuration:

All sequencer nodes share the same database so ensure that the storage configuration for each sequencer matches.
All sequencer nodes must be configured with *high-availability.enabled = true*.

Note: Starting from canton 2.4.0, sequencer high availability is enabled by default when using supported storage.

```
canton {
  sequencers {
    sequencer1 {
      sequencer {
        type = database
        high-availability.enabled = true
      }
```

The Domain node only supports embedded sequencers, so a distributed setup using a domain manager node must then be configured to use these Sequencer nodes by pointing it at these external services.

Once configured the domain must be bootstrapped with the new external sequencer using the *bootstrap_domain* operational process. These sequencers share a database so just use a single instance for bootstrapping and the replicas will come online once the shared database has sufficient state for starting.

As these nodes are likely running in separate processes you could run this command entirely externally using a remote administration configuration.

```
canton {
  remote-domains {
    da {
      # these details are provided to other nodes to use for how they should
↪connect to the embedded sequencer
      public-api {
        address = da-domain.local
        port = 1234
      }
      admin-api {
        address = da-domain.local
        port = 1235
      }
    }
  }

  remote-sequencers {
    sequencer1 {
      # these details are provided to other nodes to use for how they should
↪connect to the sequencer
      public-api {
        address = sequencer1.local
        port = 1235
      }
      # the server used from running administration commands
      admin-api {
        address = sequencer1.local
        port = 1235
      }
    }
  }
}
```

There are two methods available for exposing the horizontally scaled sequencer instances to participants.

Total Node Count

The `sequencer.high-availability.total-node-count` parameter is used to divide up time among the database sequencers. The parameter should not be changed once a set of sequencer nodes have been deployed. Because each message sequenced must have a unique timestamp, a sequencer node will use timestamps *modulo* the `total-node-count` plus own index in order to create timestamps that do not conflict with other sequencer nodes while sequencing the messages in a parallel database insertion process. Canton uses microseconds, which yields a theoretical max throughput of 1 million messages per second per domain. Now, this theoretical throughput is divided equally among all sequencer nodes (`total-node-count`). Therefore, if you set `total-node-count` too high, then a sequencer might not be able to operate at the maximum theoretical throughput. We recommend keeping the default value of `10`, as all above explanations are only theoretical and we have not yet seen a database/hard disk that can handle the theoretical throughput. Also note that a message might contain multiple events, such that we are talking about high numbers here.

External load balancer

Using a load balancer is recommended when you have a http2+grpc supporting load balancer available, and can't/don't want to expose details of the backend sequencers to clients. An advanced deployment could also support elastically scaling the number of sequencers available and dynamically reconfigure the load balancer for this updated set.

An example HAProxy configuration for exposing gRPC services without TLS looks like:

```
frontend domain_frontend
  bind 1234 proto h2
  default_backend domain_backend

backend domain_backend
  option httpchk
  http-check connect
  http-check send meth GET uri /health
  balance roundrobin
  server sequencer1 sequencer1.local:1234 proto h2 check port 8080
  server sequencer2 sequencer2.local:1234 proto h2 check port 8080
  server sequencer3 sequencer3.local:1234 proto h2 check port 8080
```

Please note that for quick failover, you also need to add HTTP health checks, as otherwise, you have to wait for the TCP timeout to occur before failover happens. The public API of the sequencer exposes the standard gRPC health endpoints, but these are currently not supported by HAProxy, hence you need to fall-back on the HTTP / health endpoint.

Client-side load balancing

Using client-side load balancing is recommended where an external load-balancing service is unavailable (or lacks http2+grpc support), and the set of sequencers is static and can be configured at the client.

To simply specify multiple sequencers use the `domains.connect_multi` console command when registering/connecting to the domain:

```
myparticipant.domains.connect_multi(
  "my_domain_alias",
  Seq("https://sequencer1.example.com", "https://sequencer2.example.com", "https:/
↪/sequencer3.example.com")
)
```

See the *sequencer connectivity documentation* for more details on how to add many sequencer urls when combined with other domain connection options. The domain connection configuration can also be changed at runtime to add or replace configured sequencer connections. Note the domain will have to be disconnected and reconnected at the participant for the updated configuration to be used.

1.25.3.5 Participant

High availability of a participant node is achieved by running multiple participant node replicas that have access to a shared database.

Participant node replicas are configured in the Canton configuration file as individual participants with two required changes for each participant node replica:

Using the same storage configuration to ensure access to the shared database. Only PostgreSQL and Oracle-based storage is supported for HA. For Oracle it is crucial that the participant replicas use the same username to access the shared database.
Set `replication.enabled = true` for each participant node replica.

Note: Starting from Canton 2.4.0, participant replication is enabled by default when using supported storage.

Domain Connectivity during Fail-over

During fail-over from one replica to another, the new active replica re-connects to all configured domains for which `manualConnect = false`. This means if the former active replica was manually connected to a domain, this domain connection is not automatically re-established during fail-over but must be performed manually again.

Manual Trigger of a Fail-over

Fail-over from the active to a passive replica is done automatically when the active replica has a failure, but one can also initiate a graceful fail-over with the following command:

```
activeParticipantReplica.replication.set_passive()
```

The command succeeds if there is at least another passive replica that takes over from the current active replica, otherwise the active replica remains active.

Load Balancer Configuration

Many replicated participants can be placed behind an appropriately sophisticated load balancer that will by health checks determine which participant instance is active and direct ledger and admin api requests to that instance appropriately. This makes participant replication and failover transparent from the perspective of the ledger-api application or canton console administering the logical participant, as they will simply be pointed at the load balancer.

Participants should be configured to expose an IsActive health status on our health HTTP server using the following monitoring configuration:

```
canton {
  monitoring {
    health {
      server {
        address = 0.0.0.0
```

(continues on next page)

(continued from previous page)

```
        port = 8000
      }

      check.type = is-active
    }
  }
}
```

Once running this server will report a HTTP 200 status code on a HTTP/1 GET request to */health* if the participant is currently the active replica. Otherwise, an error will be returned.

To use a load balancer it must support HTTP/1 health checks for routing requests on a separate HTTP/2 (gRPC) server. This is possible with HAProxy using the following example configuration:

```
global
    log stdout format raw local0

defaults
    log global
    mode http
    option httplog
    # enabled so long running connections are logged immediately upon connect
    option logasap

# expose the admin-api and ledger-api as separate servers
frontend admin-api
    bind :15001 proto h2
    default_backend admin-api

backend admin-api
    # enable http health checks
    option httpchk
    # required to create a separate connection to query the load balancer.
    # this is particularly important as the health HTTP server does not support h2
    # which would otherwise be the default.
    http-check connect
    # set the health check uri
    http-check send meth GET uri /health

    # list all participant backends
    server participant1 participant1.lan:15001 proto h2 check port 8080
    server participant2 participant2.lan:15001 proto h2 check port 8080
    server participant3 participant3.lan:15001 proto h2 check port 8080

# repeat a similar configuration to the above for the ledger-api
frontend ledger-api
    bind :15000 proto h2
    default_backend ledger-api

backend ledger-api
    option httpchk
    http-check connect
    http-check send meth GET uri /health

    server participant1 participant1.lan:15000 proto h2 check port 8080
    server participant2 participant2.lan:15000 proto h2 check port 8080
```

(continues on next page)

(continued from previous page)

```
    server participant3 participant3.lan:15000 proto h2 check port 8080
```

1.26 Disaster Recovery (DR)

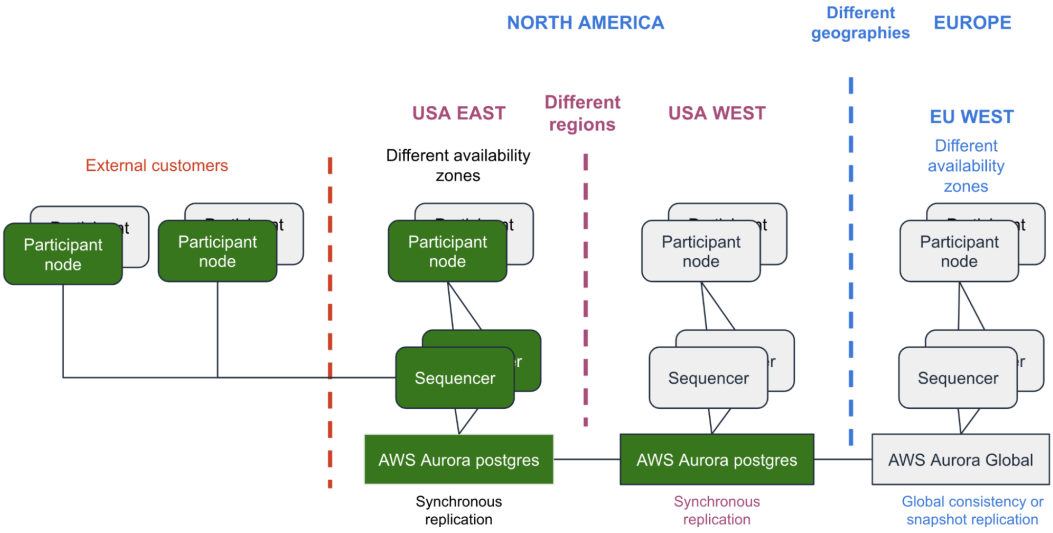
> Disaster recovery (DR) is the process of maintaining or reestablishing vital infrastructure and systems following a natural or human-induced disaster, such as a storm or battle. It employs policies, tools, and procedures.
>
> Disaster recovery assumes that the primary site is not immediately recoverable and restores data and services to a secondary site.[1]

DR is only briefly introduced here because it includes business-related recovery processes and mechanisms that are beyond the Daml solution. The previously introduced metrics of Recovery Point Objective (RTO) and Recovery Time Objective (RPO) are important. Recovery from disaster is typically measured using values for RTO and RPO.

In an Aurora global database used for DR, RTO can be in the order of minutes whereas RPO is typically measured in seconds. With an Aurora PostgreSQL-based global database, you can use the `rds.global_db_rpo` parameter to set and track the upper bound on RPO, but doing so might affect transaction processing on the primary cluster's writer node.

For more information, see the AWS documentation on managing RPOs for Aurora PostgreSQL-based global databases.

The figure below expands on the HA AWS example by adding topology which addresses DR incidents.



DR is usually more costly to architect and deploy than an HA solution. DR is expected to occur less frequently than an HA incident so the RTO for DR is longer than HA, perhaps even allowing some data loss in a DR incident.

There are different approaches to keeping the backup databases in a DR solution as synchronized as possible to an active DB. One approach is to take frequent snapshots of the source and live database(s) and send them to the remote deployment that supports DR. The AWS documentation states the following:

[1] https://en.wikipedia.org/wiki/Disaster_recovery as retrieved 02/22/2023

> You can restore a snapshot of an Aurora DB cluster or from an Amazon RDS DB instance to use as the starting point for your Aurora global database. You restore the snapshot and create a new Aurora-provisioned DB cluster at the same time. You then add another AWS Region to the restored DB cluster, thus turning it into an Aurora global database. Any Aurora DB cluster that you create using a snapshot in this way becomes the primary cluster of your Aurora global database.[2]

It's important to take care during and after a failover in a DR situation. AWS advises:

> Make sure that application writes are sent to the correct Aurora DB cluster before, during, and after making these changes. Doing this avoids data inconsistencies among the DB clusters in the Aurora global database (split-brain issues).[3]

Alternatively, AWS says in Managing RPOs for Aurora PostgreSQL-based global databases:

> With an Aurora PostgreSQL-based global database, you can manage the recovery point objective (RPO) for your Aurora global database by using PostgreSQL's `rds.global_db_rpo` parameter. RPO represents the maximum amount of data that can be lost in the event of an outage.
>
> This parameter is supported by Aurora PostgreSQL. Valid values for `rds.global_db_rpo` range from 20 seconds to 2,147,483,647 seconds (68 years).[4]

Some additional AWS links of interest on this topic:

> Fast failover with Amazon Aurora PostgreSQL.
> Fast recovery after failover with cluster cache management for Aurora PostgreSQL.

1.27 Persistence

Participant and domain nodes both require storage configurations. Both use the same configuration format and therefore support the same configuration options. There are three different configurations available:

1. `Memory` - Using simple, hash-map backed in-memory stores which are deleted whenever a node is stopped.
2. `Postgres` - To use with the open source relational database Postgres.
3. `Oracle` - To use with Oracle DB (Enterprise only)

In order to set a certain storage type, we have to edit the storage section of the particular node, such as `canton.participants.myparticipant.storage.type = memory`. Memory storage does not require any other setting.

For the actual database driver, Canton does not directly define how they are configured, but leverages a third party library (slick) for it, exposing all configuration methods therein. If you need to, please consult the respective detailed documentation to learn about all configuration options if you want to leverage any exotic option. Here, we will only describe our default, recommended and supported setup.

[2] https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-global-database-getting-started.html#aurora-global-database.use-snapshot
[3] https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-global-database-disaster-recovery.html#aurora-global-database-failover
[4] https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-global-database-disaster-recovery.html#aurora-global-database-manage-recovery

It is recommended to use a connection pool in production environments and consciously choose the size of the pool.

Please note that Canton will create, manage and upgrade the database schema directly. You don't have to create tables yourselves.

Consult the `example/03-advanced-configuration` directory to get a set of configuration files to set your nodes up.

1.27.1 Postgres

Our reference driver based definition for Postgres configuration is:

```
# Postgres persistence configuration mixin
#
# This file defines a shared configuration resources. You can mix it into your
 ↪configuration by
# refer to the shared storage resource and add the database name.
#
# Example:
#   participant1 {
#     storage = ${_shared.storage}
#     storage.config.properties.databaseName = "participant1"
#   }
#
# The user and password credentials are set to "canton" and "supersafe". As this
 ↪is not "supersafe", you might
# want to either change this configuration file or pass the settings in via
 ↪environment variables.
#
_shared {
  storage {
    type = postgres
    config {
      dataSourceClass = "org.postgresql.ds.PGSimpleDataSource"
      properties = {
        serverName = "localhost"
        # the next line will override above "serverName" in case the environment
 ↪variable POSTGRES_HOST exists
        serverName = ${?POSTGRES_HOST}
        portNumber = "5432"
        portNumber = ${?POSTGRES_PORT}
        # the next line will fail configuration parsing if the POSTGRES_USER
 ↪environment variable is not set
        user = ${POSTGRES_USER}
        password = ${POSTGRES_PASSWORD}
      }
    }
    // If defined, will configure the number of database connections per node.
    // Please ensure that your database is setup with sufficient connections.
    // If not configured explicitly, every node will create one connection per
 ↪core on the host machine. This is
    // subject to change with future improvements.
    parameters.max-connections = ${?POSTGRES_NUM_CONNECTIONS}
  }
}
```

You may use this configuration file with environment variables or adapt it accordingly. More detailed setup instructions and options are available in the Slick reference guide. The above configurations are included in the `examples/03-advanced-configuration/storage` folder and are sufficient to get going.

1.27.1.1 SSL

This snippet shows how ssl can be configured for Postgres. You can find more information about the settings in the (postgres documentation):

```
_shared {
  storage {
    type = postgres
    config {
      dataSourceClass = "org.postgresql.ds.PGSimpleDataSource"
      properties = {
        serverName = "localhost"
        serverName = ${?POSTGRES_HOST}
        portNumber = "5432"
        portNumber = ${?POSTGRES_PORT}
        user = ${POSTGRES_USER}
        password = ${POSTGRES_PASSWORD}
        # The following settings can be used to configure an SSL connection to
↪the Postgres DB
        ssl = true
        # Will verify that the server certificate is trusted
        sslmode= "verify-ca" # Other options and their meaning can be found
↪https://jdbc.postgresql.org/documentation/head/ssl-client.html

        # Optionally set with path to root certificate. Not necessary if the
↪server certificate can be verified using the JRE root certificates
        # sslrootcert = "path/to/root.cert"

        # For mTLS:
        # sslcert= "path/to/client-cert.pem"
        # sslkey= "path/to/client-key.p12"
      }
    }
  }
}
```

Note that all configuration properties for the database will be propagated to the Ledger API JDBC URL.

1.27.2 Oracle

Important: This feature is only available in Canton Enterprise

An Oracle database can be used as the local persistence for the Canton nodes. The enterprise version of Canton comes with default configuration mixins using Oracle as a database backend.

Persistence using Oracle has the following dependencies:

Oracle Database 19c - requires version 19.11 or later

Oracle Text 19c - a plugin schema to oracle database
Intel x86-64 architecture

1.27.2.1 Installation and Setup of Oracle

Assuming that Oracle has already been installed, the following configuration aspects and setup steps are required.

Default Character Set and Collations

The database must use the recommended Oracle defaults for character sets and collations:

```
AL32UTF8 encoding for NLS_CHARACTERSET
AL32UTF8 or AL16UTF16 for NLS_NCHAR_CHARACTERSET
BINARY for NLS_SORT and NLS_COMP
```

Otherwise, Canton will refuse to connect to the database and log an error message of the form

```
DatabaseConfigError(Oracle NLS database parameter ... is ..., but should be ...)
```

In addition to keeping the default database characterset and collations configurations, the Java user language must be set to `en` and the user country to `US` (the default on most systems). This can be forced by setting the `JAVA_OPTS` options via the command line additions `-Duser.language=en -Duser.country=US` (see *JVM Arguments*). Otherwise the node at startup may complain about session `NLS_SORT` or `NLS_COMP` being different from BINARY by logging these strings:

```
DatabaseConfigError(Oracle NLS session parameter NLS_SORT is ..., but
should be BINARY)
DatabaseConfigError(Oracle NLS session parameter NLS_COMP is ..., but
should be BINARY)
```

Database Replication

To allow for recovery from data loss due to catastrophic events at data centers, database replication should be enabled. The technical details of setting up replication are out of scope of this manual. Canton on Oracle assumes that a database transaction is reported as committed only after it has been persisted to all database replicas. Please make sure this is the case to prevent data corruption / data loss in case of a data center failover.

Setup Oracle Schemas

For a simple Oracle-based Canton deployment with one domain and one participant the following Oracle schemas (i.e., users) are required:

Component	Schema name	Description	Authentication
Oracle Domain	DD4ODRUN	Runtime user	Password configured per 2.2.7 Site administrator may change at will (i.e., default password is never hardcoded or assumed)
Participant	DD4OPRUN	Runtime user for Participant Canton component	
	DD4OPLEDG	Runtime user for Participant API ledger component	

The DD4ODRUN, DD4OPRUN, and DD4OPLEDG users all need the following schema privileges:

Quota Unlimited
Create table
Create type
Create session
Create view
Create procedure
Create sequence

Run the following commands as the system user (e.g., for the runtime user (DD4OPRUN) provisioning using Oracle SQL*Plus from the command line):

```
SQL> CREATE USER DD4OPRUN IDENTIFIED BY securepass;
SQL> ALTER USER DD4OPRUN QUOTA UNLIMITED ON USERS;
SQL> GRANT CREATE TABLE, CREATE TYPE, CREATE SESSION, CREATE VIEW, CREATE
 ↪PROCEDURE, CREATE SEQUENCE, CREATE TRIGGER TO DD4OPRUN;
SQL> GRANT EXECUTE ON SYS.DBMS_LOCK TO DD4OPRUN;
SQL> GRANT SELECT ON V_$MYSTAT TO DD4OPRUN;
SQL> GRANT SELECT ON V_$LOCK TO DD4OPRUN;
SQL> GRANT SELECT ON V_$PARAMETER TO DD4OPRUN;
```

For additional domain or participant nodes create the corresponding schemas with one schema per node.

If you are getting an error messages like:

```
ORA-65096: invalid common user or role name
```

you are most likely logged into the CDB instead of the PDB. Find the right PDB and change the session:

```
SQL> show pdbs
SQL> alter session SET container = ORCLPDB1;
```

You can then test whether creating the user worked using `sqlplus`:

```
sqlplus -L DD4OPRUN/securepass@ORCLPDB1
```

1.27.2.2 Configuring Canton Nodes for Oracle

The following is an example configuration for an Oracle-backed domain for the persistence of its sequencer, mediator, and topology manager nodes. The placeholders <ORACLE_HOST>, <ORACLE_PORT>, and <ORACLE_DB> will need to be replaced with the correct settings to match the environment and <ORACLE_USER> with a unique user for each node:

```
_shared {
  // Please note that this configuration only applies for domain nodes. Use
↪oracle-participant.conf to run a participant node with Oracle storage
  storage {
    type = oracle
    config {
      driver = "oracle.jdbc.OracleDriver"
      url = "jdbc:oracle:thin:@"${ORACLE_HOST}":"${ORACLE_PORT}/${ORACLE_DB}
      password = ${ORACLE_PASSWORD}
      user = ${ORACLE_USER}
    }
  }
}
```

The environment variable for ORACLE_PASSWORD needs to be set and exported so that it is accessible for substitution in the configuration files.

The persistence configuration for the Participant is an extended version based on the previous configuration for participant nodes with the addition of the Ledger API JDBC URL string:

```
include required("oracle.conf")
// note: the ledger api server (part of a canton system) requires a separate
↪schema (user) in oracle
// because of that, you need to set up a second user. here, we assume the second
↪user is set up on the same oracle db
// host using the same password as the participant schema
_shared.storage.parameters.ledger-api-jdbc-url = "jdbc:oracle:thin:"${ORACLE_USER_
↪LAPI}"/"${ORACLE_PASSWORD}"@"${ORACLE_HOST}":"${ORACLE_PORT}/${ORACLE_DB}
```

1.27.2.3 Performance Tuning

The following configuration changes serve as an example to tune the performance of Oracle. **NOTE:** The configuration changes need to be reviewed and adapted to the specific application and environment.

Operating System Modifications

Runtime Kernel Parameters

The recommended Linux kernel is version 5.10 or later. For RHEL systems, a mainline kernel can be installed from ELRepo, as follows:

```
$ sudo dnf -y install https://www.elrepo.org/elrepo-release-8.el8.elrepo.noarch.
↪rpm
$ sudo rpm --import https://www.elrepo.org/RPM-GPG-KEY-elrepo.org
$ sudo dnf makecache
$ sudo dnf --disablerepo="*" --enablerepo="elrepo-kernel" install -y kernel-ml.
↪x86_64
```

By default, the Linux kernel default settings are optimized for general-purpose applications, and as such these settings can be unsuitable or even detrimental to the performance and stability of I/O-heavy applications, like databases.

Make the following additions to `/etc/sysctl.conf`

```
vm.swappiness = 5
vm.dirty_background_ratio = 5
vm.dirty_background_bytes = 25
vm.nr_hugepages = 200
fs.file-max = 6815744
kernel.sem = 250 32000 100 128
kernel.shmmni = 4096
kernel.shmall = 1073741824
kernel.shmmax = 4398046511104
kernel.panic_on_oops = 1
net.core.rmem_default = 262144
net.core.rmem_max = 4194304
net.core.wmem_default = 262144
net.core.wmem_max = 1048576
net.ipv4.conf.all.rp_filter = 2
net.ipv4.conf.default.rp_filter = 2
fs.aio-max-nr = 1048576
net.ipv4.ip_local_port_range = 9000 65500
```

Either reboot the database server host or apply the changes to a running server by running the following command from the terminal: `sudo sysctl -p`. Upon successfully applying the new settings, `sysctl` will output the newly applied values to the console.

Shared Memory (SHM) Segments

Oracle database works best when it can keep as much working data in memory as possible, shared amongst the different subsystems, running in their own distinct OS-level processes. This memory space is used by the database System Global Area (SGA) for allocating the buffer cache pools, shared and large pools, Java process pools and stream pools, among other functions. To allocate 80% of total system memory (RAM) to the database instance, you need to allocate fractionally more system memory to the shared memory area on the OS level.

Run this command to calculate the allocation size of the SHM:

```
$ printf "%.0f\n" `echo "(\`grep MemTotal /proc/meminfo | awk '{print $2}'\`/
↪1024)*.82" | bc -s`
105712
```

Next, update `/etc/fstab` to ensure the allocation:

```
$ grep shm /etc/fstab
tmpfs       /dev/shm        tmpfs    rw,nosuid,nodev,size=105712m    0       0
```

Again, either reboot the database server host, or apply the changes to a running server by remounting the SHM tmpfs filesystem:

```
$ sudo mount -o remount /dev/shm
Verify the new settings:
$ df -h -BM -P /dev/shm
Filesystem     1048576-blocks   Used Available Capacity Mounted on
tmpfs                105712M 38912M    66800M      37% /dev/shm
```

System Container Configuration (CDB)

The System Container stores the system settings and metadata required to manage all user databases. Now modify some of the default performance settings, in multiple stages.

After each stage restart the service from within the `sqlplus` client, as follows:

```
SQL> SHUTDOWN;
SQL> STARTUP;
SQL> ALTER PLUGGABLE DATABASE ALL OPEN;
SQL> ALTER SYSTEM REGISTER;
```

Stage 1: Increase Database Memory Allocation

Allocate 80% of total available system memory to the database instance. First, calculate the value on the command line, as follows:

```
$ printf "%.0f\n" `echo "(\`grep MemTotal /proc/meminfo | awk '{print $2}'\`/
↪1024)*.8" | bc -s`
103134
```

From the database client connected to the CDB, set the memory cap, and restart the database:

```
SQL> ALTER SYSTEM SET MEMORY_TARGET = 103134M SCOPE = SPFILE;
```

Stage 2: Set Runtime Values

Also allocate 40% of total available system memory to the database's Program Global Area (PGA). The PGA is a non-shared memory region that is allocated to the CDB when the server starts. The PGA regions are also allocated per-process in the user database, and you will allocate a total amount to be used by all processes.

Again calculate the value on the command line, as follows:

```
$ printf "%.0f\n" `echo "(\`grep MemTotal /proc/meminfo | awk '{print $2}'\`/
↪1024)*.4" | bc -s`
51462
```

From the database client connected to the CDB, set the following and restart the database:

```
SQL> ALTER SYSTEM SET PGA_AGGREGATE_TARGET = 51462M SCOPE = BOTH;
SQL> ALTER SYSTEM SET RESOURCE_LIMIT = FALSE SCOPE = BOTH;
SQL> ALTER SYSTEM SET OPEN_CURSORS = 16000 SCOPE = SPFILE;
SQL> ALTER SYSTEM SET JOB_QUEUE_PROCESSES = 2000 SCOPE = BOTH;
SQL> ALTER SYSTEM SET USE_LARGE_PAGES = TRUE SCOPE = SPFILE;
SQL> ALTER SYSTEM SET SESSION_MAX_OPEN_FILES = 50 SCOPE = SPFILE;
SQL> ALTER SYSTEM SET PARALLEL_DEGREE_POLICY = AUTO SCOPE = BOTH;
SQL> ALTER SYSTEM SET DB_BIG_TABLE_CACHE_PERCENT_TARGET = 20 SCOPE = SPFILE;
SQL> ALTER SYSTEM SET DB_CACHE_SIZE = 8G SCOPE = SPFILE;
SQL> ALTER SYSTEM SET JAVA_POOL_SIZE = 8G SCOPE = SPFILE;
SQL> ALTER SYSTEM SET OPTIMIZER_ADAPTIVE_REPORTING_ONLY = TRUE SCOPE = BOTH;
SQL> ALTER SYSTEM SET OPTIMIZER_ADAPTIVE_STATISTICS = TRUE SCOPE = BOTH;
SQL> ALTER SYSTEM SET OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES = TRUE SCOPE = BOTH;
SQL> ALTER SYSTEM SET OPTIMIZER_SESSION_TYPE = ADHOC SCOPE = SPFILE;
SQL> ALTER SYSTEM SET OPTIMIZER_USE_PENDING_STATISTICS = TRUE SCOPE = BOTH;
SQL> ALTER SYSTEM SET FILESYSTEMIO_OPTIONS = SETALL SCOPE = SPFILE;
SQL> ALTER SYSTEM SET DISK_ASYNCH_IO = TRUE SCOPE = SPFILE;
SQL> ALTER SYSTEM SET PARALLEL_THREADS_PER_CPU = 8 SCOPE = BOTH;
SQL> ALTER SYSTEM SET PARALLEL_DEGREE_LIMIT = IO SCOPE = BOTH;
```

NOTE: Please avoid setting explicit process and session limits. Oracle will derive intelligent limits for you. If you still need to set explicit limits on your database, please ensure that the limits are sufficiently high for the nodes that you intend to run. We recommend configuring at least **6 times more connections per node** than actively used to give sufficient buffer for delayed connection clean-up by Oracle.. The max connection settings can be configured as explained in *Max Connection Settings*.

If your database resource limits are lower than the database connections created by the nodes, the nodes will fail to properly start or operate. If you set the number of connections too low, the system will not perform at peak throughput.

Stage 3: Configure Pluggable Database (PDB) Runtime Values

A Pluggable Database (PDB) is a user-created set of schemas, objects, and related structures that appears logically to a client application as a separate database. Do some initial configuration of the PDB to ensure it can meet the performance requirements of the your application, after which you will create new user schemas in the PDB.

If your application requires significantly larger tablespace, Oracle will resize tablespaces on-the-fly, meaning persistence grows gradually over time to fit the size requirements of the application; however, this comes at the expense of performance, as the database regularly performs blocking I/O operations to resize tablespaces, resulting in a volatile system load profile and overall reduced transaction throughput.

Overcome this limitation by pre-allocating new `TEMP`, `USERS` and `UNDO` tablespaces:

```
SQL> CREATE BIGFILE TEMPORARY TABLESPACE temp_bigfile TEMPFILE '/opt/oracle/
↪oradata/ORCLCDB/ORCLPDB1/temp_bigfile_01.dbf' SIZE 1T AUTOEXTEND ON MAXSIZE
↪UNLIMITED;
SQL> CREATE BIGFILE UNDO TABLESPACE undo_bigfile DATAFILE '/opt/oracle/oradata/
↪ORCLCDB/ORCLPDB1/undo_bigfile_01.dbf' SIZE 1T AUTOEXTEND ON MAXSIZE UNLIMITED
↪RETENTION GUARANTEE;
SQL> CREATE BIGFILE TABLESPACE users_bigfile DATAFILE '/opt/oracle/oradata/
↪ORCLCDB/ORCLPDB1/users_bigfile_01.dbf' SIZE 6T AUTOEXTEND ON MAXSIZE UNLIMITED;
```

And then reconfigure the PDB to use the new tablespaces by default:

```
SQL> ALTER DATABASE SET DEFAULT BIGFILE TABLESPACE;
SQL> ALTER DATABASE DEFAULT TEMPORARY TABLESPACE TEMP_BIGFILE;
SQL> ALTER DATABASE DEFAULT TABLESPACE USERS_BIGFILE;
SQL> ALTER SYSTEM SET UNDO_TABLESPACE = UNDO_BIGFILE SCOPE = BOTH;
```

Change the default retention to 30 minutes, giving better transaction rollback performance, after which you will restart the database:

```
SQL> ALTER SYSTEM SET UNDO_RETENTION = 1800 SCOPE = BOTH;
```

1.27.3 General Settings

1.27.3.1 Max Connection Settings

The storage configuration can further be tuned using the following additional setting:

```
canton.participants.<service-name>.storage.parameters.max-connections = X
```

This allows you to set the maximum number of DB connections used by a Canton node. If the value is None or non-positive, the value will be the number of processors. The setting has no effect if the number of connections is already set via slick options (i.e. storage.config.numThreads).

If you are unsure how to size your connection pools, this article may be a good starting point.

The number of parallel indexer connections can be configured via

```
canton.participants.<service-name>.parameters.ledgerApiServerParameters.indexer.
↪ingestion-parallelism = Y
```

A Canton participant node will establish up to `X + Y + 2` permanent connections with the database, whereas a domain node will use up to `X` permanent connections, except for a sequencer with HA setup that will allocate up to `2X` connections. During startup, the node will use an additional set of at most `X` temporary connections during database initialisation.

Please note that this number represents an upper bound of permanent connections and can be divided internally for different purposes, depending on the implementation. Consequently, the actual size of the write connection pool, for example, could be smaller.

1.27.3.2 Queue Size

Canton may schedule more database queries than the database can handle. As a result, these queries will be placed into the database queue. By default, the database queue has a size of 1000 queries. Reaching the queueing limit will lead to a `DB_STORAGE_DEGRADATION` warning. The impact of this warning is that the queuing will overflow into the asynchronous execution context and slowly degrade the processing, which will result in less database queries being created. However, for high performance setups, such spikes might occur more regularly. Therefore, to avoid the degradation warning appearing too frequent, the queue size can be configured using:

```
canton.participants.participant1.storage.config.queueSize = 10000
```

1.27.4 Backup and Restore

It is recommended that your database is frequently backed up so that the data can be restored in case of a disaster.

In the case of a restore, a participant can replay missing data from the domain as long as the domain's backup is more recent than that of the participant's.

1.27.4.1 Order of Backups

It is important that the participant's backup is not more recent than that of the domain's, as that would constitute a ledger fork. Therefore, if you back up both participant and domain databases, always back up the participant database before the domain. If you are using a domain integration, then backup the sequencer node before backing up the underlying domain storage (e.g. Besu files).

In case of a domain restore from a backup, if a participant is ahead of the domain the participant will refuse to connect to the domain (`ForkHappened`) and you must either:

restore the participant's state to a backup before the disaster of the domain, or
roll out a new domain as a repair strategy in order to *recover from a lost domain*

The state of applications that interact with participant's ledger API must be backed up before the participant, otherwise the application state has to be reset.

1.27.4.2 Restore Caveats

When restoring Canton nodes from a backup, the following caveats apply due to the loss of data between the point of backup and latest state of the nodes.

Incomplete Command Deduplication State

After the restore, the participant's in-flight submission tracking will be out of sync with what the participant has sent to the sequencer after the backup was taken. If an application resubmits a duplicate command it may get accepted even though it should have been deduplicated by the participant.

This tracking will be in sync again when:

the participant has processed all events from the sequencer, and
no queue on the sequencer includes any submission request of a transfer/transaction request from before the restore that could be sequenced again

Such submission requests have a max sequencing time of the ledger time plus the ledger-time-record-time-tolerance of the domain. It should be enough to observe a timestamp from the domain that is after the time when the participant was stopped before the restore by more than the tolerance. Once such a timestamp is observed, the in-flight submission tracking is in sync again and applications can resume submitting commands with full command deduplication guarantees.

Application State Reset

If the application's state is newer than the participant's state, either because the application was backed up after the participant or because the application is run by a different organization and wasn't restored from a backup, then the application state has to be reset. Otherwise the application has already requested and processed transactions that were lost by the participant due to the gap between when the backup was taken and when the node disaster happened.

This includes all applications that are ledger API clients of the participant, including the JSON API server.

Private Keys

Assume a scenario in which a node needs to rotate its cryptographic private key, which is currently stored in the database of the node. If the key rotation has been announced in the system before a backup has been performed, the new key will not be available on a restore, but all other nodes in the system expect the new key to be used.

To avoid this situation, perform the key rotation steps in this order:

1. Generate the new private key and store it in the database
2. Back up the database
3. Once the backup is complete, revoke the previous key

1.27.4.3 Postgres Example

If you are using Postgres to persist the participant or domain node data, you can create backups to a file and restore it using Postgres's utility commands `pg_dump` and `pg_restore` as shown below:

Backing up Postgres database to a file:

```
pg_dump -U <user> -h <host> -p <port> -w -F tar -f <fileName> <dbName>
```

Restoring Postgres database data from a file:

```
pg_restore -U <user> -h <host> -p <port> -w -d <dbName> <fileName>
```

Although the approach shown above works for small deployments, it is not recommended in larger deployments. For that, we suggest looking into incremental backups and refer to the resources below:

PostgreSQL Documentation: Backup and Restore
How incremental backups work in PostgreSQL

1.27.5 Database Replication for Disaster Recovery

1.27.5.1 Synchronous Replication

We recommend that in production at least the domain should be run with offsite synchronous replication to assure that the state of the domain is always newer than the state of the participants. However to avoid similar *caveats as with backup restore <restore_caveats>* the participants should either use synchronous replication too or as part of the manual disaster recovery failure procedure the caveats have to be addressed.

A database backup allows you to recover the ledger up to the point when the last backup was created. However, any command accepted after creation of the backup may be lost in case of a disaster. Therefore, restoring a backup will likely result in data loss.

If such data loss is unacceptable, you need to run Canton against a replicated database, which replicates its state to another site. If the original site is down due to a disaster, Canton can be started in the other site based on the replicated state in the database. It is crucial that there are no writers left in the original site to the database, because the database mechanism used in Canton to avoid multiple writers and thus avoid data corruption does not work across sites.

For detailed instructions on how to setup a replicated database and how to perform failovers, we refer to the database system documentation, e.g. the high availability documentation of PostgreSQL.

It is strongly recommended to configure replication as synchronous. That means, the database should report a database transaction as successfully committed only after it has been persisted to all database replicas. In PostgreSQL, this corresponds to the setting `synchronous_commit = on`. If you do not follow this recommendation, you may observe data loss and/or a corrupt state after a database failover. Enabling synchronous replication may impact the performance of Canton depending on the network latency between the primary and offsite database.

For PostgreSQL, Canton strives to validate the database replication configuration and fail with an error, if a misconfiguration is detected. However, this validation is of a best-effort nature; so it may fail to detect an incorrect replication configuration. For Oracle, no attempt is made to validate the database configuration. Overall, you should not rely on Canton detecting mistakes in the database configuration.

1.28 Canton Administration Quickstart

1.28.1 Command-line Arguments

Canton supports a variety of command line arguments. Please run `bin/canton --help` to see all of them. Here, we explain the most relevant ones.

1.28.1.1 Selecting a Configuration

Canton requires a configuration file to run. There is no default topology configuration built in and therefore, the user needs to at least define what kind of node (domain or participant) and how many they want to run in the given process. Sample configuration files can be found in our release package, under the `examples` directory.

When starting Canton, configuration files can be provided using

```
bin/canton --config conf_filename -c conf_filename2
```

which will start Canton by merging the content of `conf_filename2` into `conf_filename`. Both options `-c` and `--config` are equivalent. If several configuration files assign values to the same key, the *last* value is taken. The section on *static configuration* explains how to write a configuration file.

You can also specify config parameters on the command line, alone or along with configuration files, to specify missing parameters or to overwrite others. This can be useful for providing simple short config info. Config parameters can be provided using `-C`:

```
bin/canton --config conf_filename -C canton.participants.participant1.storage.
↪type=memory
```

1.28.1.2 Run Modes

Canton can run in three different modes, depending on the desired environment and task.

Interactive Console

The default method to run Canton is in the interactive mode. The process will start *a command line interface* (REPL) which allows to conveniently operate, modify and inspect the Canton application.

In this mode, all errors will be reported as `CommandExecutionException` to the console, but Canton will remain running.

The interactive console can be started together with a script, using the `--boostrap-script=...` option. The script uses the same syntax as the console.

The interactive mode is useful for development, education and expert use.

Daemon

If the console is undesired such as in server operation, Canton can be started in daemon mode

```
bin/canton daemon --config ...
```

All configured entities will be automatically started and will resume operation.

A failure to connect to the database storage will lead the process to exit with a non-zero exit code. This can be turned off using:

```
canton.participants.participant1.storage.parameters.fail-fast-on-startup = "no"
```

Any failures encountered while running the bootstrap script will immediately shutdown the Canton process with a non-zero exit code.

Nodes started in daemon mode can be administrated by setting up a *remote console* that provides the interactive user experience, while the nodes run in a separate process.

Headless Script Mode

For testing and scripting purposes, Canton can also start in headless script mode:

```
bin/canton run <script-path> --config ...
```

In this case, commands are specified in a script rather than executed interactively. Any errors with the script or during command execution should cause the Canton process to exit with a non-zero exit code.

Interactive Server Process using Screen

In some situations, we find it convenient to run even a server process interactively. For server use on Linux / OSX, this can be accomplished by using the screen command:

```
screen -S canton -d -m ./bin/canton -c ...
```

will start the Canton process in a screen session named `canton` which does not terminate on user-logout and therefore allows to inspect the Canton process whenever necessary.

A previously started process can be joined using

```
screen -r canton
```

and an active screen session can be detached using CTRL-A + D (in sequence). Be careful and avoid typing CTRL-D, as it will terminate the session. The screen session will continue to run even if you log out of the machine.

1.28.1.3 Java Virtual Machine Arguments

The `bin/canton` application is a convenient wrapper to start a Java virtual machine running the Canton process. The wrapper supports providing additional JVM options using the `JAVA_OPTS` environment variable or using the `-D` command line option.

For example, you can configure the heap size as follows:

```
JAVA_OPTS="-Xmx2G" ./bin/canton --config ...
```

There are several log related options that can be specified. Refer to *Logging* for more details.

1.28.2 Canton Console

Canton offers a console (REPL) where entities can be dynamically started and stopped, and a variety of administrative or debugging commands can be run.

All console commands must be valid Scala (the console is built on Ammonite - a Scala based scripting and REPL framework). Note that we also define a set of implicit type conversions to improve the console usability: notably, whenever a console command requires a DomainAlias, Fingerprint or Identifier, you can instead also call it with a `String` which will be automatically converted to the correct type (i.e., you can, e.g., write `participant1.domains.get_agreement("domain1")` instead of `participant1.domains.get_agreement(DomainAlias.tryCreate("domain1"))`).

The `examples/` sub-directories contain some sample scripts, with the extension `.canton`.

Contents

Commands are organised by thematic groups. Some commands also need to be explicitly turned on via configuration directives to be accessible.

Some operations are available on both types of nodes, whereas some operations are specific to either participant or domain nodes. For consistency, we organise the manual by node type, which means that some commands will appear twice. However, the detailed explanations are only given within the participant documentation.

1.28.2.1 Remote Administration

The console works in-process against local nodes. However, you can also run the console separate from the node process, and you can use a single console to administrate many remote nodes.

As an example, you might start Canton in daemon mode using

```
./bin/canton daemon -c <some config>
```

Assuming now that you've started a participant, you can access this participant using a `remote-participant` configuration such as:

```
canton {
  remote-participants {
    remoteParticipant1 {
      admin-api {
        port = 10012
        address = 127.0.0.1 // is the default value if omitted
      }
      ledger-api {
        port = 10011
```

(continues on next page)

(continued from previous page)

```
        address = 127.0.0.1 // is the default value if omitted
      }
    }
  }
}
```

Naturally, you can then also use the remote configuration to run a script:

```
./bin/canton daemon -c remote-participant1.conf --bootstrap <some-script>
```

Please note that a remote node will support almost all commands except a few that a local node supports.

If you want to generate a skeleton remote configuration of a normal config file, you can use

```
./bin/canton generate remote-config -c participant1.conf
```

However, you might have then to edit the config and adjust the hostname.

For production use cases, in particular if the Admin Api is not just bound to localhost, we recommend to enable *TLS* with mutual authentication.

1.28.2.2 Node References

To issue the command on a particular node, you must refer to it via its reference, which is a Scala variable. Named variables are created for all domain entities and participants using their configured identifiers. For example the sample `examples/01-simple-topology/simple-topology.conf` configuration file references the domain `mydomain`, and participants `participant1` and `participant2`. These are available in the console as `mydomain`, `participant1` and `participant2`.

The console also provides additional generic references that allow you to consult a list of nodes by type. The generic node reference supports three subsets of each node type: local, remote or all nodes of that type. For the participants, you can use:

```
participants.local
participants.remote
participants.all
```

The generic node references can be used in a Scala syntactic way:

```
participants.all.foreach(_.dars.upload("my.dar"))
```

but the participant references also support some *generic commands* for actions that often have to be performed for many nodes at once, such as:

```
participants.local.dars.upload("my.dar")
```

The available node references are:

<console-topic-marker: Generic Node References>

1.28.2.3 Help

Canton can be very helpful if you ask for help. Try to type

```
help
```

or

```
participant1.help()
```

to get an overview of the commands and command groups that exist. `help()` works on every level (e.g. `participant1.domains.help()`) or can be used to search for particular functions (`help("list")`) or to get detailed help explanation for each command (`participant1.parties.help("list")`).

1.28.2.4 Lifecycle Operations

These are supported by individual and sequences of domains and participants. If called on a sequence, operations will be called sequentially in the order of the sequence. For example:

```
nodes.local.start()
```

can be used to start all configured local domains and participants.

If the node is running with database persistence, it will support the database migration command (`db.migrate`). The migrations are performed automatically when the node is started for the first time. However, new migrations added as part of new versions of the software must be by default run manually using the command. In some rare cases, it may also be necessary to run `db.repair_migration` before running `db.migrate` - please refer to the description of `db.repair_migration` for more details. If desired, the database migrations can be performed also automatically by enabling the migrate-and-start mode using the following configuration option:

```
canton.participants.participant1.storage.parameters.migrate-and-start = yes
```

Note that data continuity (and therefore database migration) is only guaranteed to work across minor and patch version updates.

The domain, sequencer and mediator nodes might need extra setup to be fully functional. Check *domain bootstrapping* for more details.

1.28.2.5 Timeouts

Console command timeouts can be configured using the respective console command timeout section in the configuration file:

```
canton.parameters.timeouts.console = {
    bounded = 2.minutes
    unbounded = Inf // infinity
    ledger-command = 2.minutes
    ping = 30.seconds
}
```

The `bounded` argument is used for all commands that should finish once processing has completed, whereas the `unbounded` timeout is used for commands where we do not control the processing time. This is used in particular for potentially very long running commands.

Some commands have specific timeout arguments that can be passed explicitly as type `NonNegativeDuration`. For convenience, the console includes by default the implicits of `scala.concurrent.duration._` and an implicit conversion from the Scala type `scala.concurrent.duration.FiniteDuration` to `NonNegativeDuration`. As a result, you can use normal Scala expressions and write timeouts as

```
participant1.health.ping(participant1, timeout = 10.seconds)
```

while the implicit conversion will take care of converting it to the right types.

Generally, there is no need to re-configure the timeouts and we recommend to just use the safe default values.

1.28.2.6 Code-Generation in Console

The Daml SDK provides code-generation utilities which create **Java** or **Scala** bindings for Daml models. These bindings are a convenient way to interact with the ledger from the console in a typed fashion. The linked documentation explains how to create these bindings using the `daml` command. The **Scala** bindings are not officially supported, so should not be used for application development.

Once you have successfully built the bindings, you can then load the resulting `jar` into the Canton console using the magic **Ammonite** import trick within console scripts:

```
interp.load.cp(os.Path("codegen.jar", base = os.pwd))

@ // the at triggers the compilation such that we can use the imports subsequently

import ...
```

1.28.2.7 Canton Administration APIs

Canton provides the *console* as a builtin mode for administrative interaction. However, under the hood, all administrative console actions are effected using the administration gRPC API. Therefore, it is also possible to write your own administration application and connect it to the administration gRPC endpoints.

The `protobuf/` sub-directories in the release artifacts contain the gRPC underlying protocol buffers. In particular, the administrative gRPC APIs are located within the `admin` sub-directories.

For example, the Ping Pong Service which implements a simple workflow to smoke-test a deployment is defined with the protocol buffer `*/protobuf/*/admin/*/ping_pong_service.proto` (where `*` denotes intermediary directories). This service is then used by the console command *health.ping*.

The protocol buffers are also available within the repository following a similar sub-directory structure as mentioned.

1.28.3 Console Commands

1.28.3.1 Top-level Commands

The following commands are available for convenience:

exit

Summary: Leave the console

help

Summary: Help with console commands; type help(<command>) for detailed help for <command>

health.dump

Summary: Generate and write a health dump of Canton's state for a bug report
Arguments:
- `outputFile`: better.files.File
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration
- `chunkSize`: Option[Int]

Return type:
- String

Description: Gathers information about the current Canton process and/or remote nodes if using the console with a remote config. The outputFile argument can be used to write the health dump to a specific path. The timeout argument can be increased when retrieving large health dumps from remote nodes. The chunkSize argument controls the size of the byte chunks streamed back from remote nodes. This can be used if encountering errors due to gRPC max inbound message size being too low.

health.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

console.command_timeout

Summary: Yields the timeout for running console commands
Return type:
- com.digitalasset.canton.config.NonNegativeDuration

Description: Yields the timeout for running console commands. When the timeout has elapsed, the console stops waiting for the command result. The command will continue running in the background.

console.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

console.set_command_timeout

Summary: Sets the timeout for running console commands.
Arguments:
- `newTimeout`: com.digitalasset.canton.config.NonNegativeDuration

Description: Sets the timeout for running console commands. When the timeout has elapsed, the console stops waiting for the command result. The command will continue running in the background. The new timeout must be positive.

logging.get_level

Summary: Determine current logging level
Arguments:
- `loggerName`: String

Return type:
- Option[ch.qos.logback.classic.Level]

logging.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

logging.last_error_trace

Summary: Returns log events for an error with the same trace-id
Arguments:
- `traceId`: String

Return type:
- Seq[String]

logging.last_errors

Summary: Returns the last errors (trace-id -> error event) that have been logged locally
Return type:
- Map[String,String]

logging.set_level

Summary: Dynamically change log level (TRACE, DEBUG, INFO, WARN, ERROR, OFF, null)
Arguments:
- `loggerName`: String
- `level`: String

utils.auto_close (Testing)

Summary: Register *AutoCloseable* object to be shutdown if Canton is shut down
Arguments:
- `closeable`: AutoCloseable

utils.contract_data_to_instance

Summary: Convert contract data to a contract instance.
Arguments:
- `contractData`: com.digitalasset.canton.admin.api.client.commands.LedgerApiTypeWrappers.ContractData
- `ledgerTime`: java.time.Instant

Return type:
- com.digitalasset.canton.protocol.SerializableContract

Description: The *utils.contract_data_to_instance* bridges the gap between *participant.ledger_api.acs* commands that return various pieces of contract data and the *participant.repair.add* command used to add contract instances as part of repair workflows. Such workflows (for example migrating contracts from other Daml ledgers to Canton participants) typically consist of extracting contract data using *participant.ledger_api.acs* commands, modifying the contract data, and then converting the *contractData* using this function before finally adding the resulting contract instances to Canton participants via *participant.repair.add*. Obtain the *contractData* by invoking *.toContractData* on the *WrappedCreatedEvent* returned by the corresponding *participant.ledger_api.acs.of_party* or *of_all* call. The *ledgerTime* parameter should be chosen to be a time meaningful to the domain

on which you plan to subsequently invoke *participant.repair.add* on and will be retained alongside the contract instance by the *participant.repair.add* invocation.

utils.contract_instance_to_data

Summary: Convert a contract instance to contract data.

Arguments:

- `contract`: com.digitalasset.canton.protocol.SerializableContract

Return type:

- com.digitalasset.canton.admin.api.client.commands.LedgerApiTypeWrappers.ContractData

Description: The *utils.contract_instance_to_data* converts a Canton contract instance to contract data , a format more amenable to inspection and modification as part of repair workflows. This function consumes the output of the *participant.testing* commands and can thus be employed in workflows geared at verifying the contents of contracts for diagnostic purposes and in environments in which the features.enable-testing-commands configuration can be (at least temporarily) enabled.

utils.generate_daml_script_participants_conf

Summary: Create a participants config for Daml script

Arguments:

- `file`: Option[String]
- `useParticipantAlias`: Boolean
- `defaultParticipant`: Option[com.digitalasset.canton.console.ParticipantReference]

Return type:

- java.io.File

Description: The generated config can be passed to *daml script* via the *participant-config* parameter. More information about the file format can be found in the documentation: It takes three arguments: - file (default to participant-config.json) - useParticipantAlias (default to true): participant aliases are used instead of UIDs - defaultParticipant (default to None): adds a default participant if provided

utils.generate_navigator_conf

Summary: Create a navigator ui-backend.conf for a participant

Arguments:

- `participant`: com.digitalasset.canton.console.LocalParticipantReference
- `file`: Option[String]

Return type:

- java.io.File

utils.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

utils.object_args

Summary: Reflective inspection of object arguments, handy to inspect case class objects

Arguments:

- `obj`: T

Return type:

- List[String]

Description: Return the list field names of the given object. Helpful function when inspecting the return result.

utils.read_all_messages_from_file

Summary: Reads several Protobuf messages from a file.
Arguments:
- `fileName`: String

Return type:
- Seq[A]

Description: Fails with an exception, if the file can't be read or parsed.

utils.read_byte_string_from_file

Summary: Reads a ByteString from a file.
Arguments:
- `fileName`: String

Return type:
- com.google.protobuf.ByteString

Description: Fails with an exception, if the file can't be read.

utils.read_first_message_from_file

Summary: Reads a single Protobuf message from a file.
Arguments:
- `fileName`: String

Return type:
- A

Description: Fails with an exception, if the file can't be read or parsed.

utils.retry_until_true

Summary: Wait for a condition to become true
Arguments:
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration
- `maxWaitPeriod`: com.digitalasset.canton.config.NonNegativeDuration
- `condition`: => Boolean
- `failure`: => String

Return type:
- (condition: => Boolean, failure: => String): Unit

Description: Wait *timeout* duration until *condition* becomes true. Retry evaluating *condition* with an exponentially increasing back-off up to *maxWaitPeriod* duration between retries.

utils.retry_until_true

Summary: Wait for a condition to become true, using default timeouts
Arguments:
- `condition`: => Boolean

Description: Wait until condition becomes true, with a timeout taken from the parameters.timeouts.console.bounded configuration parameter.

utils.synchronize_topology

Summary: Wait until all topology changes have been effected on all accessible nodes
Arguments:
- `timeoutO`: Option[com.digitalasset.canton.config.NonNegativeDuration]

utils.type_args

Summary: Reflective inspection of type arguments, handy to inspect case class types
Return type:
- List[String]

Description: Return the list of field names of the given type. Helpful function when creating new objects for requests.

utils.write_to_file

Summary: Writes a ByteString to a file.
Arguments:
- `data`: com.google.protobuf.ByteString
- `fileName`: String

utils.write_to_file

Summary: Writes a Protobuf message to a file.
Arguments:
- `data`: scalapb.GeneratedMessage
- `fileName`: String

utils.write_to_file

Summary: Writes several Protobuf messages to a file.
Arguments:
- `data`: Seq[scalapb.GeneratedMessage]
- `fileName`: String

ledger_api_utils.create (Testing)

Summary: Build create command
Arguments:
- `packageId`: String
- `module`: String
- `template`: String
- `arguments`: Map[String,Any]

Return type:
- com.daml.ledger.api.v1.commands.Command

ledger_api_utils.exercise (Testing)

Summary: Build exercise command from CreatedEvent
Arguments:
- `choice`: String
- `arguments`: Map[String,Any]
- `event`: com.daml.ledger.api.v1.event.CreatedEvent

Return type:
- com.daml.ledger.api.v1.commands.Command

ledger_api_utils.exercise (Testing)

Summary: Build exercise command
Arguments:
- `packageId`: String
- `module`: String
- `template`: String
- `choice`: String
- `arguments`: Map[String,Any]
- `contractId`: String

Return type:
- com.daml.ledger.api.v1.commands.Command

ledger_api_utils.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

1.28.3.2 Participant Commands

clear_cache (Testing)
: **Summary**: Clear locally cached variables
Description: Some commands cache values on the client side. Use this command to explicitly clear the caches of these values.

config
: **Summary**: Return participant config
Return type:
- com.digitalasset.canton.participant.config.LocalParticipantConfig

help
: **Summary**: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

id
: **Summary**: Yields the globally unique id of this participant. Throws an exception, if the id has not yet been allocated (e.g., the participant has not yet been started).
Return type:
- com.digitalasset.canton.topology.ParticipantId

is_initialized
: **Summary**: Check if the local instance is running and is fully initialized
Return type:
- Boolean

is_running
: **Summary**: Check if the local instance is running
Return type:
- Boolean

start
: **Summary**: Start the instance

stop
: **Summary**: Stop the instance

testing.acs_search (Testing)
: **Summary**: Lookup of active contracts
Arguments:
- `domainAlias`: com.digitalasset.canton.DomainAlias
- `filterId`: String
- `filterPackage`: String
- `filterTemplate`: String
- `filterStakeholder`: Option[com.digitalasset.canton.topology.PartyId]
- `limit`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:
- List[com.digitalasset.canton.protocol.SerializableContract]

testing.await_domain_time (Testing)
: **Summary**: Await for the given time to be reached on the given domain
Arguments:
- `domainId`: com.digitalasset.canton.topology.DomainId

- `time`: com.digitalasset.canton.data.CantonTimestamp
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration

testing.await_domain_time (Testing)

Summary: Await for the given time to be reached on the given domain

Arguments:

- `domainAlias`: com.digitalasset.canton.DomainAlias
- `time`: com.digitalasset.canton.data.CantonTimestamp
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration

testing.bong (Testing)

Summary: Send a bong to a set of target parties over the ledger. Levels > 0 leads to an exploding ping with exponential number of contracts. Throw a RuntimeException in case of failure.

Arguments:

- `targets`: Set[com.digitalasset.canton.topology.ParticipantId]
- `validators`: Set[com.digitalasset.canton.topology.ParticipantId]
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration
- `levels`: Long
- `gracePeriodMillis`: Long
- `workflowId`: String
- `id`: String

Return type:

- scala.concurrent.duration.Duration

Description: Initiates a racy ping to multiple participants, measuring the roundtrip time of the fastest responder, with an optional timeout. Grace-period is the time the bong will wait for a duplicate spent (which would indicate an error in the system) before exiting. If levels > 0, the ping command will lead to a binary explosion and subsequent dilation of contracts, where `level` determines the number of levels we will explode. As a result, the system will create (2^(L+2) - 3) contracts (where L stands for `level`). Normally, only the initiator is a validator. Additional validators can be added using the validators argument. The bong command comes handy to run a burst test against the system and quickly leads to an overloading state.

testing.crypto_api (Testing)

Summary: Return the sync crypto api provider, which provides access to all cryptographic methods

Return type:

- com.digitalasset.canton.crypto.SyncCryptoApiProvider

testing.event_search (Testing)

Summary: Lookup of events

Arguments:

- `domain`: com.digitalasset.canton.DomainAlias
- `from`: Option[java.time.Instant]
- `to`: Option[java.time.Instant]
- `limit`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:

- Seq[(String, com.digitalasset.canton.participant.sync.TimestampedEvent)]

Description: Show the event logs. To select only events from a particular domain, use the domain alias. Leave the domain blank to search the combined event log containing the events of all domains. Note that if the domain is left blank, the values of *from* and *to* cannot be set. This is because the combined event log isn't guaranteed to have increasing

timestamps.

testing.fetch_domain_time (Testing)

Summary: Fetch the current time from the given domain

Arguments:

- `domainId`: com.digitalasset.canton.topology.DomainId
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration

Return type:

- com.digitalasset.canton.data.CantonTimestamp

testing.fetch_domain_time (Testing)

Summary: Fetch the current time from the given domain

Arguments:

- `domainAlias`: com.digitalasset.canton.DomainAlias
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration

Return type:

- com.digitalasset.canton.data.CantonTimestamp

testing.fetch_domain_times (Testing)

Summary: Fetch the current time from all connected domains

Arguments:

- `timeout`: com.digitalasset.canton.config.NonNegativeDuration

testing.find_clean_commitments_timestamp (Testing)

Summary: The latest timestamp before or at the given one for which no commitment is outstanding

Arguments:

- `domain`: com.digitalasset.canton.DomainAlias
- `beforeOrAt`: com.digitalasset.canton.data.CantonTimestamp

Return type:

- Option[com.digitalasset.canton.data.CantonTimestamp]

Description: The latest timestamp before or at the given one for which no commitment is outstanding. Note that this doesn't imply that pruning is possible at this timestamp, as the system might require some additional data for crash recovery. Thus, this is useful for testing commitments; use the commands in the pruning group for pruning. Additionally, the result needn't fall on a commitment tick as specified by the reconciliation interval.

testing.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

testing.maybe_bong (Testing)

Summary: Like bong, but returns None in case of failure.

Arguments:

- `targets`: Set[com.digitalasset.canton.topology.ParticipantId]
- `validators`: Set[com.digitalasset.canton.topology.ParticipantId]
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration
- `levels`: Long
- `gracePeriodMillis`: Long
- `workflowId`: String
- `id`: String

Return type:

- Option[scala.concurrent.duration.Duration]

testing.pcs_search (Testing)

Summary: Lookup contracts in the Private Contract Store

Arguments:

- `domainAlias`: com.digitalasset.canton.DomainAlias
- `filterId`: String
- `filterPackage`: String
- `filterTemplate`: String
- `activeSet`: Boolean
- `limit`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:

- List[(Boolean, com.digitalasset.canton.protocol.SerializableContract)]

Description: Get raw access to the PCS of the given domain sync controller. The filter commands will check if the target value `contains` the given string. The arguments can be started with `^` such that `startsWith` is used for comparison or `!` to use `equals`. The `activeSet` argument allows to restrict the search to the active contract set.

testing.sequencer_messages (Testing)

Summary: Retrieve all sequencer messages

Arguments:

- `domain`: com.digitalasset.canton.DomainAlias
- `from`: Option[java.time.Instant]
- `to`: Option[java.time.Instant]
- `limit`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:

- Seq[com.digitalasset.canton.sequencing.PossiblyIgnoredProtocolEvent]

Description: Optionally allows filtering for sequencer from a certain time span (inclusive on both ends) and limiting the number of displayed messages. The returned messages will be ordered on most domain ledger implementations if a time span is given. Fails if the participant has never connected to the domain.

testing.state_inspection (Testing)

Summary: Obtain access to the state inspection interface. Use at your own risk.

Return type:

- com.digitalasset.canton.participant.admin.SyncStateInspection

Description: The state inspection methods can fatally and permanently corrupt the state of a participant. The API is subject to change in any way.

testing.transaction_search (Testing)

Summary: Lookup of accepted transactions

Arguments:

- `domain`: com.digitalasset.canton.DomainAlias
- `from`: Option[java.time.Instant]
- `to`: Option[java.time.Instant]
- `limit`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:

- Seq[(String, com.digitalasset.canton.protocol.LfCommittedTransaction)]

Description: Show the accepted transactions as they appear in the event logs. To select only transactions from a particular domain, use the domain alias. Leave the domain blank to search the combined event log containing the events of all domains. Note that if the domain is left blank, the values of *from* and *to* cannot be set. This is because the combined event log isn't guaranteed to have increasing timestamps.

Database

db.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

db.migrate

Summary: Migrates the instance's database if using a database storage

db.repair_migration

Summary: Only use when advised - repairs the database migration of the instance's database
Arguments:
- `force`: Boolean

Description: In some rare cases, we change already applied database migration files in a new release and the repair command resets the checksums we use to ensure that in general already applied migration files have not been changed. You should only use *db.repair_migration* when advised and otherwise use it at your own risk - in the worst case running it may lead to data corruption when an incompatible database migration (one that should be rejected because the already applied database migration files have changed) is subsequently falsely applied.

Health

health.active

Summary: Check if the node is running and is the active instance (mediator, participant)
Return type:
- Boolean

health.dump

Summary: Creates a zip file containing diagnostic information about the canton process running this node
Arguments:
- `outputFile`: better.files.File
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration
- `chunkSize`: Option[Int]

Return type:
- String

health.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

health.initialized

Summary: Returns true if node has been initialized.
Return type:
- Boolean

health.maybe_ping (Testing)

Summary: Sends a ping to the target participant over the ledger. Yields Some(duration) in case of success and None in case of failure.
Arguments:
- `participantId`: com.digitalasset.canton.topology.ParticipantId
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration
- `workflowId`: String
- `id`: String

Return type:
- Option[scala.concurrent.duration.Duration]

health.ping

Summary: Sends a ping to the target participant over the ledger. Yields the duration in case of success and throws a RuntimeException in case of failure.
Arguments:
- `participantId`: com.digitalasset.canton.topology.ParticipantId
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration
- `workflowId`: String
- `id`: String

Return type:
- scala.concurrent.duration.Duration

health.running

Summary: Check if the node is running
Return type:
- Boolean

health.status

Summary: Get human (and machine) readable status info
Return type:
- com.digitalasset.canton.health.admin.data.NodeStatus[S]

health.wait_for_identity

Summary: Wait for the node to have an identity
Description: This is specifically useful for the Domain Manager which needs its identity to be ready for bootstrapping, but for which we can't rely on wait_for_initialized() because it will be initialized only after being bootstrapped.

health.wait_for_initialized

Summary: Wait for the node to be initialized

health.wait_for_running

Summary: Wait for the node to be running

Domain Connectivity

domains.accept_agreement

Summary: Accept the service agreement of the given domain alias
Arguments:
- `domainAlias`: com.digitalasset.canton.DomainAlias
- `agreementId`: String

domains.active

Summary: Test whether a participant is connected to and permissioned on a domain reference, both from the perspective of the participant and the domain.

Arguments:
- `reference`: com.digitalasset.canton.console.commands.DomainAdministration

Return type:
- Boolean

Description: Yields false, if the domain has not been initialized, is not connected or is not healthy.

domains.active

Summary: Test whether a participant is connected to and permissioned on a domain.

Arguments:
- `domainAlias`: com.digitalasset.canton.DomainAlias

Return type:
- Boolean

Description: Yields false, if the domain is not connected or not healthy. Yields false, if the domain is configured in the Canton configuration and the participant is not active from the perspective of the domain.

domains.config

Summary: Returns the current configuration of a given domain

Arguments:
- `domain`: com.digitalasset.canton.DomainAlias

Return type:
- Option[com.digitalasset.canton.participant.domain.DomainConnectionConfig]

domains.connect

Summary: Macro to connect a participant to a domain given by connection

Arguments:
- `domainAlias`: com.digitalasset.canton.DomainAlias
- `connection`: String
- `manualConnect`: Boolean
- `domainId`: Option[com.digitalasset.canton.topology.DomainId]
- `certificatesPath`: String
- `priority`: Int
- `timeTrackerConfig`: com.digitalasset.canton.config.DomainTimeTrackerConfig
- `synchronize`: Option[com.digitalasset.canton.config.NonNegativeDuration]

Return type:
- com.digitalasset.canton.participant.domain.DomainConnectionConfig

Description: The connect macro performs a series of commands in order to connect this participant to a domain. First, *register* will be invoked with the given arguments, but first registered with manualConnect = true. If you already set manualConnect = true, then nothing else will happen and you will have to do the remaining steps yourselves. Otherwise, if the domain requires an agreement, it is fetched and presented to the user for evaluation. If the user is fine with it, the agreement is confirmed. If you want to auto-confirm, then set the environment variable CANTON_AUTO_APPROVE_AGREEMENTS=yes. Finally, the command will invoke *reconnect* to startup the connection. If the reconnect succeeded, the registered configuration will be updated with manualStart = true. If anything fails, the domain will remain registered with *manualConnect = true* and you will have to perform these steps manually. The arguments are: domainAlias - The name you will be using to refer to this domain. Can not be changed anymore. connection - The connection string to connect to this domain. I.e. https://url:port manualConnect - Whether this connection should be handled manually and also excluded from automatic re-connect. domainId - Optionally the domainId you expect to see on this domain. certificatesPath - Path to TLS certificate files to use as a trust anchor. priority - The priority of the domain. The higher the

more likely a domain will be used. timeTrackerConfig - The configuration for the domain time tracker. synchronize - A timeout duration indicating how long to wait for all topology changes to have been effected on all local nodes.

domains.connect

Summary: Macro to connect a participant to a domain given by connection

Arguments:

- `config`: com.digitalasset.canton.participant.domain.DomainConnectionConfig

Description: This variant of connect expects a domain connection config. Otherwise the behaviour is equivalent to the connect command with explicit arguments. If the domain is already configured, the domain connection will be attempted. If however the domain is offline, the command will fail. Generally, this macro should only be used to setup a new domain. However, for convenience, we support idempotent invocations where subsequent calls just ensure that the participant reconnects to the domain.

domains.connect_ha

Summary: Deprecated macro to connect a participant to a domain that supports connecting via many endpoints

Arguments:

- `domainAlias`: com.digitalasset.canton.DomainAlias
- `firstConnection`: com.digitalasset.canton.sequencing.SequencerConnection
- `additionalConnections`: com.digitalasset.canton.sequencing.SequencerConnection*

Return type:

- com.digitalasset.canton.participant.domain.DomainConnectionConfig

Description: Use the command connect_ha with the updated arguments list

domains.connect_local

Summary: Macro to connect a participant to a locally configured domain given by reference

Arguments:

- `domain`: com.digitalasset.canton.console.InstanceReferenceWithSequencerConnection
- `manualConnect`: Boolean
- `alias`: Option[com.digitalasset.canton.DomainAlias]
- `maxRetryDelayMillis`: Option[Long]
- `priority`: Int
- `synchronize`: Option[com.digitalasset.canton.config.NonNegativeDuration]

Description: The arguments are: domain - A local domain or sequencer reference manualConnect - Whether this connection should be handled manually and also excluded from automatic re-connect. alias - The name you will be using to refer to this domain. Can not be changed anymore. certificatesPath - Path to TLS certificate files to use as a trust anchor. priority - The priority of the domain. The higher the more likely a domain will be used. synchronize - A timeout duration indicating how long to wait for all topology changes to have been effected on all local nodes.

domains.connect_multi

Summary: Macro to connect a participant to a domain that supports connecting via many endpoints

Arguments:

- `domainAlias`: com.digitalasset.canton.DomainAlias
- `connections`: Seq[com.digitalasset.canton.sequencing.SequencerConnection]
- `synchronize`: Option[com.digitalasset.canton.config.NonNegativeDuration]

Return type:
- com.digitalasset.canton.participant.domain.DomainConnectionConfig

Description: Domains can provide many endpoints to connect to for availability and performance benefits. This version of connect allows specifying multiple endpoints for a single domain connection: connect_multi(mydomain , Seq(sequencer1, sequencer2)) or: connect_multi(mydomain , Seq(https://host1.mydomain.net , https://host2.mydomain.net , https://host3.mydomain.net)) To create a more advanced connection config use domains.toConfig with a single host, then use config.addConnection to add additional connections before connecting: config = myparticipaint.domains.toConfig(mydomain , https://host1.mydomain.net , otherArguments) config = config.addConnection(https://host2.mydomain.net , https://host3.mydomain.net) myparticipant.domains.connect(config) The arguments are: domainAlias - The name you will be using to refer to this domain. Can not be changed anymore. connections - The sequencer connection definitions (can be an URL) to connect to this domain. I.e. https://url:port synchronize - A timeout duration indicating how long to wait for all topology changes to have been effected on all local nodes.

domains.disconnect

Summary: Disconnect this participant from the given domain

Arguments:
- `domainAlias`: com.digitalasset.canton.DomainAlias

domains.disconnect_all

Summary: Disconnect this participant from all connected domains

domains.disconnect_local

Summary: Disconnect this participant from the given local domain

Arguments:
- `domain`: com.digitalasset.canton.console.DomainReference

domains.get_agreement

Summary: Get the service agreement of the given domain alias and if it has been accepted already.

Arguments:
- `domainAlias`: com.digitalasset.canton.DomainAlias

Return type:
- Option[(com.digitalasset.canton.participant.admin.v0.Agreement, Boolean)]

domains.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:
- `methodName`: String

domains.id_of

Summary: Returns the id of the given domain alias

Arguments:
- `domainAlias`: com.digitalasset.canton.DomainAlias

Return type:
- com.digitalasset.canton.topology.DomainId

domains.is_connected

Summary: Test whether a participant is connected to a domain reference

Arguments:
- `reference`: com.digitalasset.canton.console.commands.DomainAdministration

Return type:
- Boolean

domains.is_registered

Summary: Returns true if a domain is registered using the given alias
Arguments:
- `domain`: com.digitalasset.canton.DomainAlias

Return type:
- Boolean

domains.list_connected

Summary: List the connected domains of this participant
Return type:
- Seq[com.digitalasset.canton.admin.api.client.data.ListConnectedDomainsResult]

domains.list_registered

Summary: List the configured domains of this participant
Return type:
- Seq[(com.digitalasset.canton.participant.domain.DomainConnectionConfig, Boolean)]

domains.modify

Summary: Modify existing domain connection
Arguments:
- `domain`: com.digitalasset.canton.DomainAlias
- `modifier`: com.digitalasset.canton.participant.domain.DomainConnectionConfig => com.digitalasset.canton.participant.domain.DomainConnectionConfig

domains.reconnect

Summary: Reconnect this participant to the given domain
Arguments:
- `domainAlias`: com.digitalasset.canton.DomainAlias
- `retry`: Boolean
- `synchronize`: Option[com.digitalasset.canton.config.NonNegativeDuration]

Return type:
- Boolean

Description: Idempotent attempts to re-establish a connection to a certain domain. If retry is set to false, the command will throw an exception if unsuccessful. If retry is set to true, the command will terminate after the first attempt with the result, but the server will keep on retrying to connect to the domain. The arguments are: domainAlias - The name you will be using to refer to this domain. Can not be changed anymore. retry - Whether the reconnect should keep on retrying until it succeeded or abort noisly if the connection attempt fails. synchronize - A timeout duration indicating how long to wait for all topology changes to have been effected on all local nodes.

domains.reconnect_all

Summary: Reconnect this participant to all domains which are not marked as manual start
Arguments:
- `ignoreFailures`: Boolean
- `synchronize`: Option[com.digitalasset.canton.config.NonNegativeDuration]

Description: The arguments are: ignoreFailures - If set to true (default), we'll attempt to connect to all, ignoring any failure synchronize - A timeout duration indicating how long to wait for all topology changes to have been effected on all local nodes.

domains.reconnect_local

Summary: Reconnect this participant to the given local domain
Arguments:
- `ref`: com.digitalasset.canton.console.DomainReference
- `retry`: Boolean
- `synchronize`: Option[com.digitalasset.canton.config.NonNegativeDuration]

Return type:
- Boolean

Description: Idempotent attempts to re-establish a connection to the given local domain. Same behaviour as generic reconnect. The arguments are: ref - The domain reference to connect to retry - Whether the reconnect should keep on retrying until it succeeded or abort noisly if the connection attempt fails. synchronize - A timeout duration indicating how long to wait for all topology changes to have been effected on all local nodes.

domains.register

Summary: Register new domain connection
Arguments:
- `config`: com.digitalasset.canton.participant.domain.DomainConnectionConfig

Description: When connecting to a domain, we need to register the domain connection and eventually accept the terms of service of the domain before we can connect. The registration process is therefore a subset of the operation. Therefore, register is equivalent to connect if the domain does not require a service agreement. However, you would usually call register only in advanced scripts.

Packages

packages.find

Summary: Find packages that contain a module with the given name
Arguments:
- `moduleName`: String
- `limitPackages`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:
- Seq[com.digitalasset.canton.participant.admin.v0.PackageDescription]

packages.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

packages.list

Summary: List packages stored on the participant
Arguments:
- `limit`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:
- Seq[com.digitalasset.canton.participant.admin.v0.PackageDescription]

Description: Supported arguments: limit - Limit on the number of packages returned (defaults to canton.parameters.console.default-limit)

packages.list_contents

Summary: List package contents
Arguments:

- `packageId`: String

Return type:

- Seq[com.digitalasset.canton.participant.admin.v0.ModuleDescription]

packages.remove (Preview)

Summary: Remove the package from Canton's package store.

Arguments:

- `packageId`: String
- `force`: Boolean

Description: The standard operation of this command checks that a package is unused and unvetted, and if so removes the package. The force flag can be used to disable the checks, but do not use the force flag unless you're certain you know what you're doing.

packages.synchronize_vetting

Summary: Ensure that all vetting transactions issued by this participant have been observed by all configured participants

Arguments:

- `timeout`: com.digitalasset.canton.config.NonNegativeDuration

Description: Sometimes, when scripting tests and demos, a dar or package is uploaded and we need to ensure that commands are only submitted once the package vetting has been observed by some other connected participant known to the console. This command can be used in such cases.

DAR Management

dars.download

Summary: Downloads the DAR file with the given hash to the given directory

Arguments:

- `darHash`: String
- `directory`: String

dars.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

dars.list

Summary: List installed DAR files

Arguments:

- `limit`: com.digitalasset.canton.config.RequireTypes.PositiveInt
- `filterName`: String

Return type:

- Seq[com.digitalasset.canton.participant.admin.v0.DarDescription]

Description: List DARs installed on this participant The arguments are: filterName: filter by name (source description) limit: Limit number of results (default none)

dars.list_contents

Summary: List contents of DAR files

Arguments:

- `hash`: String

Return type:

- com.digitalasset.canton.admin.api.client.data.DarMetadata

dars.remove (Preview)

Summary: Remove a DAR from the participant
Arguments:
- `darHash`: String
- `synchronizeVetting`: Boolean

Description: Can be used to remove a DAR from the participant, if the following conditions are satisfied: 1. The main package of the DAR must be unused – there should be no active contract from this package 2. All package dependencies of the DAR should either be unused or contained in another of the participant node's uploaded DARs. Canton uses this restriction to ensure that the package dependencies of the DAR don't become stranded if they're in use. 3. The main package of the dar should not be vetted. If it is vetted, Canton will try to automatically revoke the vetting for the main package of the DAR, but this automatic vetting revocation will only succeed if the main package vetting originates from a standard `dars.upload`. Even if the automatic revocation fails, you can always manually revoke the package vetting. If synchronizeVetting is true (default), then the command will block until the participant has observed the vetting transactions to be registered with the domain.

dars.upload

Summary: Upload a Dar to Canton
Arguments:
- `path`: String
- `vetAllPackages`: Boolean
- `synchronizeVetting`: Boolean

Return type:
- String

Description: Daml code is normally shipped as a Dar archive and must explicitly be uploaded to a participant. A Dar is a collection of LF-packages, the native binary representation of Daml smart contracts. In order to use Daml templates on a participant, the Dar must first be uploaded and then vetted by the participant. Vetting will ensure that other participants can check whether they can actually send a transaction referring to a particular Daml package and participant. Vetting is done by registering a VettedPackages topology transaction with the topology manager. By default, vetting happens automatically and this command waits for the vetting transaction to be successfully registered on all connected domains. This is the safe default setting minimizing race conditions. If vetAllPackages is true (default), the packages will all be vetted on all domains the participant is registered. If synchronizeVetting is true (default), then the command will block until the participant has observed the vetting transactions to be registered with the domain. Note that synchronize vetting might block on permissioned domains that do not just allow participants to update the topology state. In such cases, synchronizeVetting should be turned off. Synchronize vetting can be invoked manually using $participant.package.synchronize_vettings()

DAR Sharing

dars.sharing.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

dars.sharing.requests.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

dars.sharing.requests.list (Preview)

Summary: List pending requests to share a DAR with others

Return type:

- Seq[com.digitalasset.canton.participant.admin.v0.ListShareRequestsResponse.Item]

dars.sharing.requests.propose (Preview)

Summary: Share a DAR with other participants

Arguments:

- `darHash`: String
- `participantId`: com.digitalasset.canton.topology.ParticipantId

dars.sharing.offers.accept (Preview)

Summary: Accept the offer to share a DAR

Arguments:

- `shareId`: String

dars.sharing.offers.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

dars.sharing.offers.list

Summary: List received DAR sharing offers

Return type:

- Seq[com.digitalasset.canton.participant.admin.v0.ListShareOffersResponse.Item]

dars.sharing.offers.reject (Preview)

Summary: Reject the offer to share a DAR

Arguments:

- `shareId`: String
- `reason`: String

dars.sharing.whitelist.add (Preview)

Summary: Add party to my DAR sharing whitelist

Arguments:

- `partyId`: com.digitalasset.canton.topology.PartyId

dars.sharing.whitelist.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

dars.sharing.whitelist.list (Preview)
Summary: List parties that are currently whitelisted to share DARs with me

dars.sharing.whitelist.remove (Preview)
Summary: Remove party from my DAR sharing whitelist
Arguments:
- `partyId`: com.digitalasset.canton.topology.PartyId

Party Management

The party management commands allow to conveniently enable and disable parties on the local node. Under the hood, they use the more complicated but feature-richer identity management commands.

parties.await_topology_observed (Preview)
Summary: Waits for any topology changes to be observed
Arguments:
- `partyAssignment`: Set[(com.digitalasset.canton.topology.PartyId, T)]
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration

Description: Will throw an exception if the given topology has not been observed within the given timeout.

parties.disable
Summary: Disable party on participant
Arguments:
- `name`: com.digitalasset.canton.topology.Identifier
- `force`: Boolean

parties.enable
Summary: Enable/add party to participant
Arguments:
- `name`: String
- `displayName`: Option[String]
- `waitForDomain`: com.digitalasset.canton.console.commands.DomainChoice
- `synchronizeParticipants`: Seq[com.digitalasset.canton.console.ParticipantReference]

Return type:
- com.digitalasset.canton.topology.PartyId

Description: This function registers a new party with the current participant within the participants namespace. The function fails if the participant does not have appropriate signing keys to issue the corresponding PartyToParticipant topology transaction. Optionally, a local display name can be added. This display name will be exposed on the ledger API party management endpoint. Specifying a set of domains via the *WaitForDomain* parameter ensures that the domains have enabled/added a party by the time the call returns, but other participants connected to the same domains may not yet be aware of the party. Additionally, a sequence of additional participants can be added to be synchronized to ensure that the party is known to these participants as well before the function terminates.

parties.find

Summary: Find a party from a filter string
Arguments:
- `filterParty`: String

Return type:
- com.digitalasset.canton.topology.PartyId

Description: Will search for all parties that match this filter string. If it finds exactly one party, it will return that one. Otherwise, the function will throw.

parties.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

parties.hosted

Summary: List parties hosted by this participant
Arguments:
- `filterParty`: String
- `filterDomain`: String
- `asOf`: Option[java.time.Instant]
- `limit`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:
- Seq[com.digitalasset.canton.admin.api.client.data.ListPartiesResult]

Description: Inspect the parties hosted by this participant as used for synchronisation. The response is built from the timestamped topology transactions of each domain, excluding the authorized store of the given node. The search will include all hosted parties and is equivalent to running the *list* method using the participant id of the invoking participant. filterParty: Filter by parties starting with the given string. filterDomain: Filter by domains whose id starts with the given string. asOf: Optional timestamp to inspect the topology state at a given point in time. limit: How many items to return (defaults to canton.parameters.console.default-limit) Example: participant1.parties.hosted(filterParty= alice)

parties.list

Summary: List active parties, their active participants, and the participants' permissions on domains.
Arguments:
- `filterParty`: String
- `filterParticipant`: String
- `filterDomain`: String
- `asOf`: Option[java.time.Instant]
- `limit`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:
- Seq[com.digitalasset.canton.admin.api.client.data.ListPartiesResult]

Description: Inspect the parties known by this participant as used for synchronisation. The response is built from the timestamped topology transactions of each domain, excluding the authorized store of the given node. For each known party, the list of active participants and their permission on the domain for that party is given. filterParty: Filter by parties starting with the given string. filterParticipant: Filter for parties that are hosted by a participant with an id starting with the given string filterDomain: Filter by domains whose id starts with the given string. asOf: Optional timestamp to inspect the topology state at a given point in time. limit: Limit on the number of parties fetched (de-

faults to canton.parameters.console.default-limit). Example: participant1.parties.list(filterParty= alice)

parties.set_display_name

Summary: Set party display name

Arguments:

- `party`: com.digitalasset.canton.topology.PartyId
- `displayName`: String

Description: Locally set the party display name (shown on the ledger-api) to the given value

parties.update

Summary: Update participant-local party details

Arguments:

- `party`: com.digitalasset.canton.topology.PartyId
- `modifier`: com.digitalasset.canton.admin.api.client.data.PartyDetails => com.digitalasset.canton.admin.api.client.data.PartyDetails

Return type:

- com.digitalasset.canton.admin.api.client.data.PartyDetails

Description: Currently you can update only the annotations. You cannot update other user attributes. party: party to be updated, modifier: a function to modify the party details, e.g.: *partyDetails => { partyDetails.copy(annotations = partyDetails.annotations.updated(a , b).removed(c)) }*

Key Administration

keys.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

keys.public.download

Summary: Download public key

Arguments:

- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint
- `protocolVersion`: com.digitalasset.canton.version.ProtocolVersion

Return type:

- com.google.protobuf.ByteString

keys.public.download_to

Summary: Download public key and save it to a file

Arguments:

- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint
- `outputFile`: String
- `protocolVersion`: com.digitalasset.canton.version.ProtocolVersion

keys.public.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

keys.public.list

Summary: List public keys in registry
Arguments:
- `filterFingerprint`: String
- `filterContext`: String

Return type:
- Seq[com.digitalasset.canton.crypto.PublicKeyWithName]

Description: Returns all public keys that have been added to the key registry. Optional arguments can be used for filtering.

keys.public.list_by_owner

Summary: List keys for given keyOwner.
Arguments:
- `keyOwner`: com.digitalasset.canton.topology.KeyOwner
- `filterDomain`: String
- `asOf`: Option[java.time.Instant]
- `limit`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:
- Seq[com.digitalasset.canton.admin.api.client.data.ListKeyOwnersResult]

Description: This command is a convenience wrapper for *list_key_owners*, taking an explicit keyOwner as search argument. The response includes the public keys.

keys.public.list_owners

Summary: List active owners with keys for given search arguments.
Arguments:
- `filterKeyOwnerUid`: String
- `filterKeyOwnerType`: Option[com.digitalasset.canton.topology.KeyOwnerCode]
- `filterDomain`: String
- `asOf`: Option[java.time.Instant]
- `limit`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:
- Seq[com.digitalasset.canton.admin.api.client.data.ListKeyOwnersResult]

Description: This command allows deep inspection of the topology state. The response includes the public keys. Optional filterKeyOwnerType type can be 'ParticipantId.Code' , 'MediatorId.Code','SequencerId.Code', 'DomainTopologyManagerId.Code'.

keys.public.upload

Summary: Upload public key
Arguments:
- `filename`: String
- `name`: Option[String]

Return type:
- com.digitalasset.canton.crypto.Fingerprint

keys.public.upload

Summary: Upload public key
Arguments:
- `keyBytes`: com.google.protobuf.ByteString
- `name`: Option[String]

Return type:
- com.digitalasset.canton.crypto.Fingerprint

Description: Import a public key and store it together with a name used to provide some

context to that key.

keys.secret.delete

Summary: Delete private key
Arguments:
- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint
- `force`: Boolean

keys.secret.download

Summary: Download key pair
Arguments:
- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint
- `protocolVersion`: com.digitalasset.canton.version.ProtocolVersion

Return type:
- com.google.protobuf.ByteString

keys.secret.download_to

Summary: Download key pair and save it to a file
Arguments:
- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint
- `outputFile`: String
- `protocolVersion`: com.digitalasset.canton.version.ProtocolVersion

keys.secret.generate_encryption_key

Summary: Generate new public/private key pair for encryption and store it in the vault
Arguments:
- `name`: String
- `scheme`: Option[com.digitalasset.canton.crypto.EncryptionKeyScheme]

Return type:
- com.digitalasset.canton.crypto.EncryptionPublicKey

Description: The optional name argument allows you to store an associated string for your convenience. The scheme can be used to select a key scheme and the default scheme is used if left unspecified.

keys.secret.generate_signing_key

Summary: Generate new public/private key pair for signing and store it in the vault
Arguments:
- `name`: String
- `scheme`: Option[com.digitalasset.canton.crypto.SigningKeyScheme]

Return type:
- com.digitalasset.canton.crypto.SigningPublicKey

Description: The optional name argument allows you to store an associated string for your convenience. The scheme can be used to select a key scheme and the default scheme is used if left unspecified.

keys.secret.get_wrapper_key_id

Summary: Get the wrapper key id that is used for the encrypted private keys store
Return type:
- String

keys.secret.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

keys.secret.list

Summary: List keys in private vault

Arguments:

- `filterFingerprint`: String
- `filterName`: String
- `purpose`: Set[com.digitalasset.canton.crypto.KeyPurpose]

Return type:

- Seq[com.digitalasset.canton.crypto.admin.grpc.PrivateKeyMetadata]

Description: Returns all public keys to the corresponding private keys in the key vault. Optional arguments can be used for filtering.

keys.secret.register_kms_encryption_key

Summary: Register the specified KMS encryption key in canton storing its public information in the vault

Arguments:

- `kmsKeyId`: String
- `name`: String

Return type:

- com.digitalasset.canton.crypto.EncryptionPublicKey

Description: The id for the KMS encryption key. The optional name argument allows you to store an associated string for your convenience.

keys.secret.register_kms_signing_key

Summary: Register the specified KMS signing key in canton storing its public information in the vault

Arguments:

- `kmsKeyId`: String
- `name`: String

Return type:

- com.digitalasset.canton.crypto.SigningPublicKey

Description: The id for the KMS signing key. The optional name argument allows you to store an associated string for your convenience.

keys.secret.rotate_kms_node_key

Summary: Rotate a given node's keypair with a new pre-generated KMS keypair

Arguments:

- `fingerprint`: String
- `newKmsKeyId`: String

Return type:

- com.digitalasset.canton.crypto.PublicKey

Description: Rotates an existing encryption or signing key stored externally in a KMS with a pre-generated key. The fingerprint of the key we want to rotate. The id of the new KMS key (e.g. Resource Name).

keys.secret.rotate_node_key

Summary: Rotate a node's public/private key pair

Arguments:

- `fingerprint`: String
- `name`: Option[String]

Return type:

- com.digitalasset.canton.crypto.PublicKey

Description: Rotates an existing encryption or signing key. NOTE: A namespace root or intermediate signing key CANNOT be rotated by this command. The fingerprint of the key

we want to rotate.

keys.secret.rotate_node_keys

Summary: Rotate the node's public/private key pairs
Description: For a participant node it rotates the signing and encryption key pair. For a domain or domain manager node it rotates the signing key pair as those nodes do not have an encryption key pair. For a sequencer or mediator node use *rotate_node_keys* with a domain manager reference as an argument. NOTE: Namespace root or intermediate signing keys are NOT rotated by this command.

keys.secret.rotate_wrapper_key

Summary: Change the wrapper key for encrypted private keys store
Arguments:
- `newWrapperKeyId`: String

Description: Change the wrapper key (e.g. AWS KMS key) being used to encrypt the private keys in the store. newWrapperKeyId: The optional new wrapper key id to be used. If the wrapper key id is empty Canton will generate a new key based on the current configuration.

keys.secret.upload

Summary: Upload a key pair
Arguments:
- `pairBytes`: com.google.protobuf.ByteString
- `name`: Option[String]

keys.secret.upload

Summary: Upload (load and import) a key pair from file
Arguments:
- `filename`: String
- `name`: Option[String]

Topology Administration

The topology commands can be used to manipulate and inspect the topology state. In all commands, we use fingerprints to refer to public keys. Internally, these fingerprints are resolved using the key registry (which is a map of Fingerprint -> PublicKey). Any key can be added to the key registry using the `keys.public.load` commands.

topology.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

topology.init_id

Summary: Initialize the node with a unique identifier
Arguments:
- `identifier`: com.digitalasset.canton.topology.Identifier
- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint

Return type:
- com.digitalasset.canton.topology.UniqueIdentifier

Description: Every node in Canton is identified using a unique identifier, which is composed of a user-chosen string and the fingerprint of a signing key. The signing key is the root key defining a so-called namespace, where the signing key has the ultimate control

over issuing new identifiers. During initialisation, we have to pick such a unique identifier. By default, initialisation happens automatically, but it can be turned off by setting the auto-init option to false. Automatic node initialisation is usually turned off to preserve the identity of a participant or domain node (during major version upgrades) or if the topology transactions are managed through a different topology manager than the one integrated into this node.

topology.load_transaction

Summary: Upload signed topology transaction

Arguments:

- `bytes`: com.google.protobuf.ByteString

Description: Topology transactions can be issued with any topology manager. In some cases, such transactions need to be copied manually between nodes. This function allows for uploading previously exported topology transaction into the authorized store (which is the name of the topology managers transaction store.

topology.stores.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

topology.stores.list

Summary: List available topology stores

Return type:

- Seq[String]

Description: Topology transactions are stored in these stores. There are the following stores: Authorized - The authorized store is the store of a topology manager. Updates to the topology state are made by adding new transactions to the Authorized store. Both the participant and the domain nodes topology manager have such a store. A participant node will distribute all the content in the Authorized store to the domains it is connected to. The domain node will distribute the content of the Authorized store through the sequencer to the domain members in order to create the authoritative topology state on a domain (which is stored in the store named using the domain-id), such that every domain member will have the same view on the topology state on a particular domain. <domain-id> - The domain store is the authorized topology state on a domain. A participant has one store for each domain it is connected to. The domain has exactly one store with its domain-id. Requested - A domain can be configured such that when participant tries to register a topology transaction with the domain, the transaction is placed into the Requested store such that it can be analysed and processed with user defined process.

topology.namespace_delegations.authorize

Summary: Change namespace delegation

Arguments:

- `ops`: com.digitalasset.canton.topology.transaction.TopologyChangeOp
- `namespace`: com.digitalasset.canton.crypto.Fingerprint
- `authorizedKey`: com.digitalasset.canton.crypto.Fingerprint
- `isRootDelegation`: Boolean
- `signedBy`: Option[com.digitalasset.canton.crypto.Fingerprint]
- `synchronize`: Option[com.digitalasset.canton.config.NonNegativeDuration]
- `force`: Boolean

Return type:

- com.google.protobuf.ByteString

Description: Delegates the authority to authorize topology transactions in a certain namespace to a certain key. The keys are referred to using their fingerprints. They need to be either locally generated or have been previously imported. ops: Either Add or Remove the delegation. namespace: The namespace whose authorization authority is delegated. signedBy: Optional fingerprint of the authorizing key. The authorizing key needs to be either the authorizedKey for root certificates. Otherwise, the signedBy key needs to refer to a previously authorized key, which means that we use the signedBy key to refer to a locally available CA. authorizedKey: Fingerprint of the key to be authorized. If signedBy equals authorizedKey, then this transaction corresponds to a self-signed root certificate. If the keys differ, then we get an intermediate CA. isRootDelegation: If set to true (default = false), the authorized key will be allowed to issue NamespaceDelegations. synchronize: Synchronize timeout can be used to ensure that the state has been propagated into the node

topology.namespace_delegations.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

topology.namespace_delegations.list

Summary: List namespace delegation transactions

Arguments:

- `filterStore`: String
- `useStateStore`: Boolean
- `timeQuery`: com.digitalasset.canton.topology.store.TimeQuery
- `operation`: Option[com.digitalasset.canton.topology.transaction.TopologyChangeOp]
- `filterNamespace`: String
- `filterSigningKey`: String
- `filterTargetKey`: Option[com.digitalasset.canton.crypto.Fingerprint]
- `protocolVersion`: Option[String]

Return type:

- Seq[com.digitalasset.canton.admin.api.client.data.ListNamespaceDelegationResult]

Description: List the namespace delegation transaction present in the stores. Namespace delegations are topology transactions that permission a key to issue topology transactions within a certain namespace. filterStore: Filter for topology stores starting with the given filter string (Authorized, <domain-id>, Requested) useStateStore: If true (default), only properly authorized transactions that are part of the state will be selected. timeQuery: The time query allows to customize the query by time. The following options are supported: TimeQuery.HeadState (default): The most recent known state. TimeQuery.Snapshot(ts): The state at a certain point in time. TimeQuery.Range(fromO, toO): Time-range of when the transaction was added to the store operation: Optionally, what type of operation the transaction should have. State store only has Add . filterSigningKey: Filter for transactions that are authorized with a key that starts with the given filter string. filterNamespace: Filter for namespaces starting with the given filter string. filterTargetKey: Filter for namespaces delegations for the given target key. protocolVersion: Export the topology transactions in the optional protocol version.

topology.identifier_delegations.authorize

Summary: Change identifier delegation

Arguments:

- `ops`: com.digitalasset.canton.topology.transaction.TopologyChangeOp

- `identifier`: com.digitalasset.canton.topology.UniqueIdentifier
- `authorizedKey`: com.digitalasset.canton.crypto.Fingerprint
- `signedBy`: Option[com.digitalasset.canton.crypto.Fingerprint]
- `synchronize`: Option[com.digitalasset.canton.config.NonNegativeDuration]

Return type:

- com.google.protobuf.ByteString

Description: Delegates the authority of a certain identifier to a certain key. This corresponds to a normal certificate which binds identifier to a key. The keys are referred to using their fingerprints. They need to be either locally generated or have been previously imported. ops: Either Add or Remove the delegation. signedBy: Refers to the optional fingerprint of the authorizing key which in turn refers to a specific, locally existing certificate. authorizedKey: Fingerprint of the key to be authorized. synchronize: Synchronize timeout can be used to ensure that the state has been propagated into the node

topology.identifier_delegations.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

topology.identifier_delegations.list

Summary: List identifier delegation transactions

Arguments:

- `filterStore`: String
- `useStateStore`: Boolean
- `timeQuery`: com.digitalasset.canton.topology.store.TimeQuery
- `operation`: Option[com.digitalasset.canton.topology.transaction.TopologyChangeOp]
- `filterUid`: String
- `filterSigningKey`: String
- `filterTargetKey`: Option[com.digitalasset.canton.crypto.Fingerprint]
- `protocolVersion`: Option[String]

Return type:

- Seq[com.digitalasset.canton.admin.api.client.data.ListIdentifierDelegationResult]

Description: List the identifier delegation transaction present in the stores. Identifier delegations are topology transactions that permission a key to issue topology transactions for a certain unique identifier. filterStore: Filter for topology stores starting with the given filter string (Authorized, <domain-id>, Requested) useStateStore: If true (default), only properly authorized transactions that are part of the state will be selected. timeQuery: The time query allows to customize the query by time. The following options are supported: TimeQuery.HeadState (default): The most recent known state. TimeQuery.Snapshot(ts): The state at a certain point in time. TimeQuery.Range(fromO, toO): Time-range of when the transaction was added to the store operation: Optionally, what type of operation the transaction should have. State store only has Add . filterSigningKey: Filter for transactions that are authorized with a key that starts with the given filter string. filterUid: Filter for unique identifiers starting with the given filter string. protocolVersion: Export the topology transactions in the optional protocol version.

topology.owner_to_key_mappings.authorize

Summary: Change an owner to key mapping

Arguments:

- `ops`: com.digitalasset.canton.topology.transaction.TopologyChangeOp

- keyOwner: com.digitalasset.canton.topology.KeyOwner
- key: com.digitalasset.canton.crypto.Fingerprint
- purpose: com.digitalasset.canton.crypto.KeyPurpose
- signedBy: Option[com.digitalasset.canton.crypto.Fingerprint]
- synchronize: Option[com.digitalasset.canton.config.NonNegativeDuration]
- force: Boolean

Return type:

- com.google.protobuf.ByteString

Description: Change a owner to key mapping. A key owner is anyone in the system that needs a key-pair known to all members (participants, mediator, sequencer, topology manager) of a domain. ops: Either Add or Remove the key mapping update. signedBy: Optional fingerprint of the authorizing key which in turn refers to a specific, locally existing certificate. ownerType: Role of the following owner (Participant, Sequencer, Mediator, DomainTopologyManager) owner: Unique identifier of the owner. key: Fingerprint of key purposes: The purposes of the owner to key mapping. force: removing the last key is dangerous and must therefore be manually forced synchronize: Synchronize timeout can be used to ensure that the state has been propagated into the node

topology.owner_to_key_mappings.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- methodName: String

topology.owner_to_key_mappings.list

Summary: List owner to key mapping transactions

Arguments:

- filterStore: String
- useStateStore: Boolean
- timeQuery: com.digitalasset.canton.topology.store.TimeQuery
- operation: Option[com.digitalasset.canton.topology.transaction.TopologyChangeOp]
- filterKeyOwnerType: Option[com.digitalasset.canton.topology.KeyOwnerCode]
- filterKeyOwnerUid: String
- filterKeyPurpose: Option[com.digitalasset.canton.crypto.KeyPurpose]
- filterSigningKey: String
- protocolVersion: Option[String]

Return type:

- Seq[com.digitalasset.canton.admin.api.client.data.ListOwnerToKeyMappingResult]

Description: List the owner to key mapping transactions present in the stores. Owner to key mappings are topology transactions defining that a certain key is used by a certain key owner. Key owners are participants, sequencers, mediators and domains. filterStore: Filter for topology stores starting with the given filter string (Authorized, <domain-id>, Requested) useStateStore: If true (default), only properly authorized transactions that are part of the state will be selected. timeQuery: The time query allows to customize the query by time. The following options are supported: TimeQuery.HeadState (default): The most recent known state. TimeQuery.Snapshot(ts): The state at a certain point in time. TimeQuery.Range(fromO, toO): Time-range of when the transaction was added to the store operation: Optionally, what type of operation the transaction should have. State store only has Add . filterSigningKey: Filter for transactions that are authorized with a key that starts

with the given filter string. filterKeyOwnerType: Filter for a particular type of key owner (KeyOwnerCode). filterKeyOwnerUid: Filter for key owners unique identifier starting with the given filter string. filterKeyPurpose: Filter for keys with a particular purpose (Encryption or Signing) protocolVersion: Export the topology transactions in the optional protocol version.

topology.owner_to_key_mappings.rotate_key

Summary: Rotate the key for an owner to key mapping

Arguments:

- `nodeInstance`: com.digitalasset.canton.console.InstanceReferenceCommon
- `owner`: com.digitalasset.canton.topology.KeyOwner
- `currentKey`: com.digitalasset.canton.crypto.PublicKey
- `newKey`: com.digitalasset.canton.crypto.PublicKey

Description: Rotates the key for an existing owner to key mapping by issuing a new owner to key mapping with the new key and removing the previous owner to key mapping with the previous key. nodeInstance: The node instance that is used to verify that both current and new key pertain to this node. This avoids conflicts when there are different nodes with the same uuid (i.e., multiple sequencers). owner: The owner of the owner to key mapping currentKey: The current public key that will be rotated newKey: The new public key that has been generated

topology.party_to_participant_mappings.authorize (Preview)

Summary: Change party to participant mapping

Arguments:

- `ops`: com.digitalasset.canton.topology.transaction.TopologyChangeOp
- `party`: com.digitalasset.canton.topology.PartyId
- `participant`: com.digitalasset.canton.topology.ParticipantId
- `side`: com.digitalasset.canton.topology.transaction.RequestSide
- `permission`: com.digitalasset.canton.topology.transaction.ParticipantPermission
- `signedBy`: Option[com.digitalasset.canton.crypto.Fingerprint]
- `synchronize`: Option[com.digitalasset.canton.config.NonNegativeDuration]
- `replaceExisting`: Boolean
- `force`: Boolean

Return type:

- com.google.protobuf.ByteString

Description: Change the association of a party to a participant. If both identifiers are in the same namespace, then the request-side is Both. If they differ, then we need to say whether the request comes from the party (RequestSide.From) or from the participant (RequestSide.To). And, we need the matching request of the other side. Please note that this is a preview feature due to the fact that inhomogeneous topologies can not yet be properly represented on the Ledger API. ops: Either Add or Remove the mapping signedBy: Refers to the optional fingerprint of the authorizing key which in turn refers to a specific, locally existing certificate. party: The unique identifier of the party we want to map to a participant. participant: The unique identifier of the participant to which the party is supposed to be mapped. side: The request side (RequestSide.From if we the transaction is from the perspective of the party, RequestSide.To from the participant.) privilege: The privilege of the given participant which allows us to restrict an association (e.g. Confirmation or Observation). replaceExisting: If true (default), replace any existing mapping with the new setting synchronize: Synchronize timeout can be used to ensure that the state has been propagated into the node

topology.party_to_participant_mappings.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

topology.party_to_participant_mappings.list

Summary: List party to participant mapping transactions
Arguments:
- `filterStore`: String
- `useStateStore`: Boolean
- `timeQuery`: com.digitalasset.canton.topology.store.TimeQuery
- `operation`: Option[com.digitalasset.canton.topology.transaction.TopologyChangeOp]
- `filterParty`: String
- `filterParticipant`: String
- `filterRequestSide`: Option[com.digitalasset.canton.topology.transaction.RequestSide]
- `filterPermission`: Option[com.digitalasset.canton.topology.transaction.ParticipantPermission]
- `filterSigningKey`: String
- `protocolVersion`: Option[String]

Return type:
- Seq[com.digitalasset.canton.admin.api.client.data.ListPartyToParticipantResult]

Description: List the party to participant mapping transactions present in the stores. Party to participant mappings are topology transactions used to allocate a party to a certain participant. The same party can be allocated on several participants with different privileges. A party to participant mapping has a request-side that identifies whether the mapping is authorized by the party, by the participant or by both. In order to have a party be allocated to a given participant, we therefore need either two transactions (one with RequestSide.From, one with RequestSide.To) or one with RequestSide.Both. filterStore: Filter for topology stores starting with the given filter string (Authorized, <domain-id>, Requested) useStateStore: If true (default), only properly authorized transactions that are part of the state will be selected. timeQuery: The time query allows to customize the query by time. The following options are supported: TimeQuery.HeadState (default): The most recent known state. TimeQuery.Snapshot(ts): The state at a certain point in time. TimeQuery.Range(fromO, toO): Time-range of when the transaction was added to the store operation: Optionally, what type of operation the transaction should have. State store only has Add . filterSigningKey: Filter for transactions that are authorized with a key that starts with the given filter string. filterParty: Filter for parties starting with the given filter string. filterParticipant: Filter for participants starting with the given filter string. filterRequestSide: Optional filter for a particular request side (Both, From, To). protocolVersion: Export the topology transactions in the optional protocol version.

topology.participant_domain_states.active

Summary: Returns true if the given participant is currently active on the given domain
Arguments:
- `domainId`: com.digitalasset.canton.topology.DomainId
- `participantId`: com.digitalasset.canton.topology.ParticipantId

Return type:
- Boolean

Description: Active means that the participant has been granted at least observation rights on the domain and that the participant has registered a domain trust certificate

topology.participant_domain_states.authorize

Summary: Change participant domain states

Arguments:

- ops: com.digitalasset.canton.topology.transaction.TopologyChangeOp
- domain: com.digitalasset.canton.topology.DomainId
- participant: com.digitalasset.canton.topology.ParticipantId
- side: com.digitalasset.canton.topology.transaction.RequestSide
- permission: com.digitalasset.canton.topology.transaction.ParticipantPermission
- trustLevel: com.digitalasset.canton.topology.transaction.TrustLevel
- signedBy: Option[com.digitalasset.canton.crypto.Fingerprint]
- synchronize: Option[com.digitalasset.canton.config.NonNegativeDuration]
- replaceExisting: Boolean

Return type:

- com.google.protobuf.ByteString

Description: Change the association of a participant to a domain. In order to activate a participant on a domain, we need both authorisation: the participant authorising its uid to be present on a particular domain and the domain to authorise the presence of a participant on said domain. If both identifiers are in the same namespace, then the request-side can be Both. If they differ, then we need to say whether the request comes from the domain (RequestSide.From) or from the participant (RequestSide.To). And, we need the matching request of the other side. ops: Either Add or Remove the mapping signedBy: Refers to the optional fingerprint of the authorizing key which in turn refers to a specific, locally existing certificate. domain: The unique identifier of the domain we want the participant to join. participant: The unique identifier of the participant. side: The request side (RequestSide.From if we the transaction is from the perspective of the domain, RequestSide.To from the participant.) permission: The privilege of the given participant which allows us to restrict an association (e.g. Confirmation or Observation). Will use the lower of if different between To/From. trustLevel: The trust level of the participant on the given domain. Will use the lower of if different between To/From. replaceExisting: If true (default), replace any existing mapping with the new setting synchronize: Synchronize timeout can be used to ensure that the state has been propagated into the node

topology.participant_domain_states.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- methodName: String

topology.participant_domain_states.list

Summary: List participant domain states

Arguments:

- filterStore: String
- useStateStore: Boolean
- timeQuery: com.digitalasset.canton.topology.store.TimeQuery
- operation: Option[com.digitalasset.canton.topology.transaction.TopologyChangeOp]
- filterDomain: String
- filterParticipant: String
- filterSigningKey: String
- protocolVersion: Option[String]

Return type:

- Seq[com.digitalasset.canton.admin.api.client.data.ListParticipantDomainStateResult]

Description: List the participant domain transactions present in the stores. Participant domain states are topology transactions used to permission a participant on a given domain. A participant domain state has a request-side that identifies whether the mapping is authorized by the participant (From), by the domain (To) or by both (Both). In order to use a participant on a domain, both have to authorize such a mapping. This means that by authorizing such a topology transaction, a participant acknowledges its presence on a domain, whereas a domain permissions the participant on that domain. filterStore: Filter for topology stores starting with the given filter string (Authorized, <domain-id>, Requested) useStateStore: If true (default), only properly authorized transactions that are part of the state will be selected. timeQuery: The time query allows to customize the query by time. The following options are supported: TimeQuery.HeadState (default): The most recent known state. TimeQuery.Snapshot(ts): The state at a certain point in time. TimeQuery.Range(fromO, toO): Time-range of when the transaction was added to the store operation: Optionally, what type of operation the transaction should have. State store only has Add . filterSigningKey: Filter for transactions that are authorized with a key that starts with the given filter string. filterDomain: Filter for domains starting with the given filter string. filterParticipant: Filter for participants starting with the given filter string. protocolVersion: Export the topology transactions in the optional protocol version.

topology.vetted_packages.authorize

Summary: Change package vettings

Arguments:

- `ops`: com.digitalasset.canton.topology.transaction.TopologyChangeOp
- `participant`: com.digitalasset.canton.topology.ParticipantId
- `packageIds`: Seq[com.daml.lf.data.Ref.PackageId]
- `signedBy`: Option[com.digitalasset.canton.crypto.Fingerprint]
- `synchronize`: Option[com.digitalasset.canton.config.NonNegativeDuration]
- `force`: Boolean

Return type:

- com.google.protobuf.ByteString

Description: A participant will only process transactions that reference packages that all involved participants have vetted previously. Vetting is done by registering a respective topology transaction with the domain, which can then be used by other participants to verify that a transaction is only using vetted packages. Note that all referenced and dependent packages must exist in the package store. By default, only vetting transactions adding new packages can be issued. Removing package vettings and issuing package vettings for other participants (if their identity is controlled through this participants topology manager) or for packages that do not exist locally can only be run using the force = true flag. However, these operations are dangerous and can lead to the situation of a participant being unable to process transactions. ops: Either Add or Remove the vetting. participant: The unique identifier of the participant that is vetting the package. packageIds: The lf-package ids to be vetted. signedBy: Refers to the fingerprint of the authorizing key which in turn must be authorized by a valid, locally existing certificate. If none is given, a key is automatically determined. synchronize: Synchronize timeout can be used to ensure that the state has been propagated into the node force: Flag to enable dangerous operations (default false). Great power requires great care.

topology.vetted_packages.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:
- `methodName`: String

topology.vetted_packages.list

Summary: List package vetting transactions
Arguments:
- `filterStore`: String
- `useStateStore`: Boolean
- `timeQuery`: com.digitalasset.canton.topology.store.TimeQuery
- `operation`: Option[com.digitalasset.canton.topology.transaction.TopologyChangeOp]
- `filterParticipant`: String
- `filterSigningKey`: String
- `protocolVersion`: Option[String]

Return type:
- Seq[com.digitalasset.canton.admin.api.client.data.ListVettedPackagesResult]

Description: List the package vetting transactions present in the stores. Participants must vet Daml packages and submitters must ensure that the receiving participants have vetted the package prior to submitting a transaction (done automatically during submission and validation). Vetting is done by authorizing such topology transactions and registering with a domain. filterStore: Filter for topology stores starting with the given filter string (Authorized, <domain-id>, Requested) useStateStore: If true (default), only properly authorized transactions that are part of the state will be selected. timeQuery: The time query allows to customize the query by time. The following options are supported: TimeQuery.HeadState (default): The most recent known state. TimeQuery.Snapshot(ts): The state at a certain point in time. TimeQuery.Range(fromO, toO): Time-range of when the transaction was added to the store operation: Optionally, what type of operation the transaction should have. State store only has Add . filterSigningKey: Filter for transactions that are authorized with a key that starts with the given filter string. filterParticipant: Filter for participants starting with the given filter string. protocolVersion: Export the topology transactions in the optional protocol version.

topology.all.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

topology.all.list

Summary: List all transaction
Arguments:
- `filterStore`: String
- `useStateStore`: Boolean
- `timeQuery`: com.digitalasset.canton.topology.store.TimeQuery
- `operation`: Option[com.digitalasset.canton.topology.transaction.TopologyChangeOp]
- `filterAuthorizedKey`: Option[com.digitalasset.canton.crypto.Fingerprint]
- `protocolVersion`: Option[String]

Return type:
- com.digitalasset.canton.topology.store.StoredTopologyTransactions[com.digitalasset.canton.topology.transaction.TopologyChangeOp]

Description: List all topology transactions in a store, independent of the particular type. This method is useful for exporting entire states. filterStore: Filter for topology stores

starting with the given filter string (Authorized, <domain-id>, Requested) useStateStore: If true (default), only properly authorized transactions that are part of the state will be selected. timeQuery: The time query allows to customize the query by time. The following options are supported: TimeQuery.HeadState (default): The most recent known state. TimeQuery.Snapshot(ts): The state at a certain point in time. TimeQuery.Range(fromO, toO): Time-range of when the transaction was added to the store operation: Optionally, what type of operation the transaction should have. State store only has Add . filterAuthorizedKey: Filter the topology transactions by the key that has authorized the transactions. protocolVersion: Export the topology transactions in the optional protocol version.

topology.all.renew

Summary: Renew all topology transactions that have been authorized with a previous key using a new key

Arguments:

- `filterAuthorizedKey`: com.digitalasset.canton.crypto.Fingerprint
- `authorizeWith`: com.digitalasset.canton.crypto.Fingerprint

Description: Finds all topology transactions that have been authorized by *filterAuthorizedKey* and renews those topology transactions by authorizing them with the new key *authorizeWith*. filterAuthorizedKey: Filter the topology transactions by the key that has authorized the transactions. authorizeWith: The key to authorize the renewed topology transactions.

Ledger API Access

The following commands on a participant reference provide access to the participant's Ledger API services.

ledger_api.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

ledger_api_v2.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

Transaction Service

ledger_api.transactions.by_id (Testing)

Summary: Get a (tree) transaction by its ID

Arguments:

- `parties`: Set[com.digitalasset.canton.topology.PartyId]
- `id`: String

Return type:

- Option[com.daml.ledger.api.v1.transaction.TransactionTree]

Description: Get a transaction tree from the transaction stream by its ID. Returns None if the transaction is not (yet) known at the participant or if the transaction has been pruned via *pruning.prune*.

ledger_api.transactions.domain_of (Testing)

Summary: Get the domain that a transaction was committed over.

Arguments:

- `transactionId`: String

Return type:

- com.digitalasset.canton.topology.DomainId

Description: Get the domain that a transaction was committed over. Throws an error if the transaction is not (yet) known to the participant or if the transaction has been pruned via *pruning.prune*.

ledger_api.transactions.end (Testing)

Summary: Get ledger end

Return type:

- com.daml.ledger.api.v1.ledger_offset.LedgerOffset

ledger_api.transactions.flat (Testing)

Summary: Get flat transactions

Arguments:

- `partyIds`: Set[com.digitalasset.canton.topology.PartyId]
- `completeAfter`: Int
- `beginOffset`: com.daml.ledger.api.v1.ledger_offset.LedgerOffset
- `endOffset`: Option[com.daml.ledger.api.v1.ledger_offset.LedgerOffset]
- `verbose`: Boolean
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration

Return type:

- Seq[com.daml.ledger.api.v1.transaction.Transaction]

Description: This function connects to the flat transaction stream for the given parties and collects transactions until either *completeAfter* transaction trees have been received or *timeout* has elapsed. The returned transactions can be filtered to be between the given offsets (default: no filtering). If the participant has been pruned via *pruning.prune* and if *beginOffset* is lower than the pruning offset, this command fails with a *NOT_FOUND* error.

ledger_api.transactions.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

ledger_api.transactions.start_measuring (Testing)

Summary: Starts measuring throughput at the transaction service

Arguments:

- `parties`: Set[com.digitalasset.canton.topology.PartyId]
- `metricSuffix`: String
- `onTransaction`: com.daml.ledger.api.v1.transaction.TransactionTree => Unit

Return type:

- AutoCloseable

Description: This function will subscribe on behalf of *parties* to the transaction tree stream and notify various metrics: The metric *<name>.<metricSuffix>* counts the number of transaction trees emitted. The metric *<name>.<metricSuffix>-tx-node-count* tracks the number of root events emitted as part of transaction trees. The metric *<name>.<metricSuffix>-tx-size* tracks the number of bytes emitted as part of transaction trees. To stop measuring, you need to close the returned *AutoCloseable*. Use the *onTransaction* parameter to register a callback that is called on every transaction tree.

ledger_api.transactions.subscribe_flat (Testing)

Summary: Subscribe to the flat transaction stream

Arguments:

- `observer`: io.grpc.stub.StreamObserver[com.daml.ledger.api.v1.transaction.Transaction]
- `filter`: com.daml.ledger.api.v1.transaction_filter.TransactionFilter
- `beginOffset`: com.daml.ledger.api.v1.ledger_offset.LedgerOffset
- `endOffset`: Option[com.daml.ledger.api.v1.ledger_offset.LedgerOffset]
- `verbose`: Boolean

Return type:

- AutoCloseable

Description: This function connects to the flat transaction stream and passes transactions to *observer* until the stream is completed. Only transactions for parties in *filter.filterByParty.keys* will be returned. Use *filter = TransactionFilter(Map(myParty.toLf -> Filters()))* to return all transactions for *myParty: PartyId*. The returned transactions can be filtered to be between the given offsets (default: no filtering). If the participant has been pruned via *pruning.prune* and if *beginOffset* is lower than the pruning offset, this command fails with a *NOT_FOUND* error.

ledger_api.transactions.subscribe_trees (Testing)

Summary: Subscribe to the transaction tree stream

Arguments:

- `observer`: io.grpc.stub.StreamObserver[com.daml.ledger.api.v1.transaction.TransactionTree]
- `filter`: com.daml.ledger.api.v1.transaction_filter.TransactionFilter
- `beginOffset`: com.daml.ledger.api.v1.ledger_offset.LedgerOffset
- `endOffset`: Option[com.daml.ledger.api.v1.ledger_offset.LedgerOffset]
- `verbose`: Boolean

Return type:

- AutoCloseable

Description: This function connects to the transaction tree stream and passes transaction trees to *observer* until the stream is completed. Only transaction trees for parties in *filter.filterByParty.keys* will be returned. Use *filter = TransactionFilter(Map(myParty.toLf -> Filters()))* to return all trees for *myParty: PartyId*. The returned transactions can be filtered to be between the given offsets (default: no filtering). If the participant has been pruned via *pruning.prune* and if *beginOffset* is lower than the pruning offset, this command fails with a *NOT_FOUND* error.

ledger_api.transactions.trees (Testing)

Summary: Get transaction trees

Arguments:

- `partyIds`: Set[com.digitalasset.canton.topology.PartyId]
- `completeAfter`: Int
- `beginOffset`: com.daml.ledger.api.v1.ledger_offset.LedgerOffset
- `endOffset`: Option[com.daml.ledger.api.v1.ledger_offset.LedgerOffset]
- `verbose`: Boolean
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration

Return type:

- Seq[com.daml.ledger.api.v1.transaction.TransactionTree]

Description: This function connects to the transaction tree stream for the given parties and collects transaction trees until either *completeAfter* transaction trees have been received or *timeout* has elapsed. The returned transaction trees can be filtered to be between

the given offsets (default: no filtering). If the participant has been pruned via *pruning.prune* and if *beginOffset* is lower than the pruning offset, this command fails with a *NOT_FOUND* error.

Command Service

ledger_api.commands.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

ledger_api.commands.submit (Testing)

Summary: Submit command and wait for the resulting transaction, returning the transaction tree or failing otherwise

Arguments:

- `actAs`: Seq[com.digitalasset.canton.topology.PartyId]
- `commands`: Seq[com.daml.ledger.api.v1.commands.Command]
- `workflowId`: String
- `commandId`: String
- `optTimeout`: Option[com.digitalasset.canton.config.NonNegativeDuration]
- `deduplicationPeriod`: Option[com.digitalasset.canton.ledger.api.DeduplicationPeriod]
- `submissionId`: String
- `minLedgerTimeAbs`: Option[java.time.Instant]
- `readAs`: Seq[com.digitalasset.canton.topology.PartyId]
- `disclosedContracts`: Seq[com.daml.ledger.api.v1.commands.DisclosedContract]
- `applicationId`: String

Return type:

- com.daml.ledger.api.v1.transaction.TransactionTree

Description: Submits a command on behalf of the *actAs* parties, waits for the resulting transaction to commit and returns it. If the timeout is set, it also waits for the transaction to appear at all other configured participants who were involved in the transaction. The call blocks until the transaction commits or fails; the timeout only specifies how long to wait at the other participants. Fails if the transaction doesn't commit, or if it doesn't become visible to the involved participants in the allotted time. Note that if the optTimeout is set and the involved parties are concurrently enabled/disabled or their participants are connected/disconnected, the command may currently result in spurious timeouts or may return before the transaction appears at all the involved participants.

ledger_api.commands.submit_async (Testing)

Summary: Submit command asynchronously

Arguments:

- `actAs`: Seq[com.digitalasset.canton.topology.PartyId]
- `commands`: Seq[com.daml.ledger.api.v1.commands.Command]
- `workflowId`: String
- `commandId`: String
- `deduplicationPeriod`: Option[com.digitalasset.canton.ledger.api.DeduplicationPeriod]
- `submissionId`: String

- `minLedgerTimeAbs`: Option[java.time.Instant]
- `readAs`: Seq[com.digitalasset.canton.topology.PartyId]
- `disclosedContracts`: Seq[com.daml.ledger.api.v1.commands.DisclosedContract]
- `applicationId`: String

Description: Provides access to the command submission service of the Ledger API. See https://docs.daml.com/app-dev/services.html for documentation of the parameters.

ledger_api.commands.submit_flat (Testing)

Summary: Submit command and wait for the resulting transaction, returning the flattened transaction or failing otherwise

Arguments:

- `actAs`: Seq[com.digitalasset.canton.topology.PartyId]
- `commands`: Seq[com.daml.ledger.api.v1.commands.Command]
- `workflowId`: String
- `commandId`: String
- `optTimeout`: Option[com.digitalasset.canton.config.NonNegativeDuration]
- `deduplicationPeriod`: Option[com.digitalasset.canton.ledger.api.DeduplicationPeriod]
- `submissionId`: String
- `minLedgerTimeAbs`: Option[java.time.Instant]
- `readAs`: Seq[com.digitalasset.canton.topology.PartyId]
- `disclosedContracts`: Seq[com.daml.ledger.api.v1.commands.DisclosedContract]
- `applicationId`: String

Return type:

- com.daml.ledger.api.v1.transaction.Transaction

Description: Submits a command on behalf of the *actAs* parties, waits for the resulting transaction to commit, and returns the flattened transaction. If the timeout is set, it also waits for the transaction to appear at all other configured participants who were involved in the transaction. The call blocks until the transaction commits or fails; the timeout only specifies how long to wait at the other participants. Fails if the transaction doesn't commit, or if it doesn't become visible to the involved participants in the allotted time. Note that if the optTimeout is set and the involved parties are concurrently enabled/disabled or their participants are connected/disconnected, the command may currently result in spurious timeouts or may return before the transaction appears at all the involved participants.

ledger_api_v2.commands.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

ledger_api_v2.commands.submit (Testing)

Summary: Submit command and wait for the resulting transaction, returning the transaction tree or failing otherwise

Arguments:

- `actAs`: Seq[com.digitalasset.canton.topology.PartyId]
- `commands`: Seq[com.daml.ledger.api.v1.commands.Command]
- `domainId`: com.digitalasset.canton.topology.DomainId
- `workflowId`: String
- `commandId`: String

- optTimeout: Option[com.digitalasset.canton.config.NonNegativeDuration]
- deduplicationPeriod: Option[com.digitalasset.canton.ledger.api.DeduplicationPeriod]
- submissionId: String
- minLedgerTimeAbs: Option[java.time.Instant]
- readAs: Seq[com.digitalasset.canton.topology.PartyId]
- disclosedContracts: Seq[com.daml.ledger.api.v1.commands.DisclosedContract]
- applicationId: String

Return type:

- com.daml.ledger.api.v2.transaction.TransactionTree

Description: Submits a command on behalf of the *actAs* parties, waits for the resulting transaction to commit and returns it. If the timeout is set, it also waits for the transaction to appear at all other configured participants who were involved in the transaction. The call blocks until the transaction commits or fails; the timeout only specifies how long to wait at the other participants. Fails if the transaction doesn't commit, or if it doesn't become visible to the involved participants in the allotted time. Note that if the optTimeout is set and the involved parties are concurrently enabled/disabled or their participants are connected/disconnected, the command may currently result in spurious timeouts or may return before the transaction appears at all the involved participants.

ledger_api_v2.commands.submit_async (Testing)

Summary: Submit command asynchronously

Arguments:

- actAs: Seq[com.digitalasset.canton.topology.PartyId]
- commands: Seq[com.daml.ledger.api.v1.commands.Command]
- domainId: com.digitalasset.canton.topology.DomainId
- workflowId: String
- commandId: String
- deduplicationPeriod: Option[com.digitalasset.canton.ledger.api.DeduplicationPeriod]
- submissionId: String
- minLedgerTimeAbs: Option[java.time.Instant]
- readAs: Seq[com.digitalasset.canton.topology.PartyId]
- disclosedContracts: Seq[com.daml.ledger.api.v1.commands.DisclosedContract]
- applicationId: String

Description: Provides access to the command submission service of the Ledger API. See https://docs.daml.com/app-dev/services.html for documentation of the parameters.

ledger_api_v2.commands.submit_flat (Testing)

Summary: Submit command and wait for the resulting transaction, returning the flattened transaction or failing otherwise

Arguments:

- actAs: Seq[com.digitalasset.canton.topology.PartyId]
- commands: Seq[com.daml.ledger.api.v1.commands.Command]
- domainId: com.digitalasset.canton.topology.DomainId
- workflowId: String
- commandId: String
- optTimeout: Option[com.digitalasset.canton.config.NonNegativeDuration]
- deduplicationPeriod: Option[com.digitalasset.canton.ledger.api.DeduplicationPeriod]

- `submissionId`: String
- `minLedgerTimeAbs`: Option[java.time.Instant]
- `readAs`: Seq[com.digitalasset.canton.topology.PartyId]
- `disclosedContracts`: Seq[com.daml.ledger.api.v1.commands.DisclosedContract]
- `applicationId`: String

Return type:

- com.daml.ledger.api.v2.transaction.Transaction

Description: Submits a command on behalf of the *actAs* parties, waits for the resulting transaction to commit, and returns the flattened transaction. If the timeout is set, it also waits for the transaction to appear at all other configured participants who were involved in the transaction. The call blocks until the transaction commits or fails; the timeout only specifies how long to wait at the other participants. Fails if the transaction doesn't commit, or if it doesn't become visible to the involved participants in the allotted time. Note that if the optTimeout is set and the involved parties are concurrently enabled/disabled or their participants are connected/disconnected, the command may currently result in spurious timeouts or may return before the transaction appears at all the involved participants.

Command Completion Service

ledger_api.completions.end (Testing)

Summary: Read the current command completion offset

Return type:

- com.daml.ledger.api.v1.ledger_offset.LedgerOffset

ledger_api.completions.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

ledger_api.completions.list (Testing)

Summary: Lists command completions following the specified offset

Arguments:

- `partyId`: com.digitalasset.canton.topology.PartyId
- `atLeastNumCompletions`: Int
- `offset`: com.daml.ledger.api.v1.ledger_offset.LedgerOffset
- `applicationId`: String
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration
- `filter`: com.daml.ledger.api.v1.completion.Completion => Boolean

Return type:

- Seq[com.daml.ledger.api.v1.completion.Completion]

Description: If the participant has been pruned via *pruning.prune* and if *offset* is lower than the pruning offset, this command fails with a *NOT_FOUND* error.

ledger_api.completions.list_with_checkpoint (Testing)

Summary: Lists command completions following the specified offset along with the checkpoints included in the completions

Arguments:

- `partyId`: com.digitalasset.canton.topology.PartyId
- `atLeastNumCompletions`: Int

- `offset`: com.daml.ledger.api.v1.ledger_offset.LedgerOffset
- `applicationId`: String
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration
- `filter`: com.daml.ledger.api.v1.completion.Completion => Boolean

Return type:

- Seq[(com.daml.ledger.api.v1.completion.Completion, Option[com.daml.ledger.api.v1.command_completion_service.Checkpoint])]

Description: If the participant has been pruned via *pruning.prune* and if *offset* is lower than the pruning offset, this command fails with a *NOT_FOUND* error.

ledger_api.completions.subscribe (Testing)

Summary: Subscribe to the command completion stream

Arguments:

- `observer`: io.grpc.stub.StreamObserver[com.daml.ledger.api.v1.completion.Completion]
- `parties`: Seq[com.digitalasset.canton.topology.PartyId]
- `beginOffset`: com.daml.ledger.api.v1.ledger_offset.LedgerOffset
- `applicationId`: String

Return type:

- AutoCloseable

Description: This function connects to the command completion stream and passes command completions to *observer* until the stream is completed. Only completions for parties in *parties* will be returned. The returned completions start at *beginOffset* (default: *LEDGER_BEGIN*). If the participant has been pruned via *pruning.prune* and if *beginOffset* is lower than the pruning offset, this command fails with a *NOT_FOUND* error.

Active Contract Service

ledger_api.acs.await (Testing)

Summary: Wait until a contract becomes available

Arguments:

- `partyId`: com.digitalasset.canton.topology.PartyId
- `companion`: com.daml.ledger.client.binding.TemplateCompanion[T]
- `predicate`: com.daml.ledger.client.binding.Contract[T] => Boolean
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration

Return type:

- (partyId: com.digitalasset.canton.topology.PartyId, companion: com.daml.ledger.client.binding.TemplateCompanion[T], predicate: com.daml.ledger.client.binding.Contract[T] => Boolean, timeout: com.digitalasset.canton.config.NonNegativeDuration): com.daml.ledger.client.binding.Contract[T]

Description: This function can be used for contracts with a code-generated Scala model. You can refine your search using the *filter* function argument. The command will wait until the contract appears or throw an exception once it times out.

ledger_api.acs.await_active_contract (Testing)

Summary: Wait until the party sees the given contract in the active contract service

Arguments:

- `party`: com.digitalasset.canton.topology.PartyId
- `contractId`: com.digitalasset.canton.protocol.LfContractId
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration

Description: Will throw an exception if the contract is not found to be active within the given timeout

ledger_api.acs.filter (Testing)

Summary: Filter the ACS for contracts of a particular Scala code-generated template

Arguments:

- `partyId`: com.digitalasset.canton.topology.PartyId
- `templateCompanion`: com.daml.ledger.client.binding.TemplateCompanion[T]
- `predicate`: com.daml.ledger.client.binding.Contract[T] => Boolean

Return type:

- (partyId: com.digitalasset.canton.topology.PartyId, templateCompanion: com.daml.ledger.client.binding.TemplateCompanion[T], predicate: com.daml.ledger.client.binding.Contract[T] => Boolean): Seq[com.daml.ledger.client.binding.Contract[T]]

Description: To use this function, ensure a code-generated Scala model for the target template exists. You can refine your search using the *predicate* function argument.

ledger_api.acs.find_generic (Testing)

Summary: Generic search for contracts

Arguments:

- `partyId`: com.digitalasset.canton.topology.PartyId
- `filter`: com.digitalasset.canton.admin.api.client.commands.LedgerApiTypeWrappers.WrappedCreatedEvent => Boolean
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration

Return type:

- com.digitalasset.canton.admin.api.client.commands.LedgerApiTypeWrappers.WrappedCreatedEvent

Description: This search function returns an untyped ledger-api event. The find will wait until the contract appears or throw an exception once it times out.

ledger_api.acs.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

ledger_api.acs.of_all (Testing)

Summary: List the set of active contracts for all parties hosted on this participant

Arguments:

- `limit`: com.digitalasset.canton.config.RequireTypes.PositiveInt
- `verbose`: Boolean
- `filterTemplates`: Seq[com.digitalasset.canton.admin.api.client.data.TemplateId]
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration
- `identityProviderId`: String

Return type:

- Seq[com.digitalasset.canton.admin.api.client.commands.LedgerApiTypeWrappers.WrappedCreatedEvent]

Description: If the filterTemplates argument is not empty, the acs lookup will filter by the given templates.

ledger_api.acs.of_party (Testing)

Summary: List the set of active contracts of a given party

Arguments:

- party: com.digitalasset.canton.topology.PartyId
- limit: com.digitalasset.canton.config.RequireTypes.PositiveInt
- verbose: Boolean
- filterTemplates: Seq[com.digitalasset.canton.admin.api.client.data.TemplateId]
- timeout: com.digitalasset.canton.config.NonNegativeDuration

Return type:

- Seq[com.digitalasset.canton.admin.api.client.commands.LedgerApiTypeWrappers.WrappedCreatedEvent]

Description: This command will return the current set of active contracts for the given party. Supported arguments: - party: for which party you want to load the acs - limit: limit (default set via canton.parameter.console) - filterTemplate: list of templates ids to filter for

Package Service

ledger_api.packages.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- methodName: String

ledger_api.packages.list (Testing)

Summary: List Daml Packages

Arguments:

- limit: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:

- Seq[com.daml.ledger.api.v1.admin.package_management_service.PackageDetails]

ledger_api.packages.upload_dar (Testing)

Summary: Upload packages from Dar file

Arguments:

- darPath: String

Description: Uploading the Dar can be done either through the ledger Api server or through the Canton admin Api. The Ledger Api is the portable method across ledgers. The Canton admin Api is more powerful as it allows for controlling Canton specific behaviour. In particular, a Dar uploaded using the ledger Api will not be available in the Dar store and can not be downloaded again. Additionally, Dars uploaded using the ledger Api will be vetted, but the system will not wait for the Dars to be successfully registered with all connected domains. As such, if a Dar is uploaded and then used immediately thereafter, a command might bounce due to missing package vettings.

Party Management Service

ledger_api.parties.allocate (Testing)

Summary: Allocate a new party
Arguments:
- `party`: String
- `displayName`: String
- `annotations`: Map[String,String]
- `identityProviderId`: String

Return type:
- com.digitalasset.canton.admin.api.client.data.PartyDetails

Description: Allocates a new party on the ledger. party: a hint for generating the party identifier displayName: a human-readable name of this party annotations: key-value pairs associated with this party and stored locally on this Ledger API server identityProviderId: identity provider id

ledger_api.parties.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

ledger_api.parties.list (Testing)

Summary: List parties known by the Ledger API server
Arguments:
- `identityProviderId`: String

Return type:
- Seq[com.digitalasset.canton.admin.api.client.data.PartyDetails]

Description: Lists parties known by the Ledger API server. identityProviderId: identity provider id

ledger_api.parties.update

Summary: Update participant-local party details
Arguments:
- `party`: com.digitalasset.canton.topology.PartyId
- `modifier`: com.digitalasset.canton.admin.api.client.data.PartyDetails => com.digitalasset.canton.admin.api.client.data.PartyDetails
- `identityProviderId`: String

Return type:
- com.digitalasset.canton.admin.api.client.data.PartyDetails

Description: Currently you can update only the annotations. You cannot update other user attributes. party: party to be updated, modifier: a function to modify the party details, e.g.: *partyDetails => { partyDetails.copy(annotations = partyDetails.annotations.updated(a , b).removed(c)) }* identityProviderId: identity provider id

ledger_api.parties.update_idp (Testing)

Summary: Update party's identity provider id
Arguments:
- `party`: com.digitalasset.canton.topology.PartyId
- `sourceIdentityProviderId`: String
- `targetIdentityProviderId`: String

Description: Updates party's identity provider id. party: party to be updated sourceIdentityProviderId: source identity provider id targetIdentityProviderId: target identity provider

id

Ledger Configuration Service

ledger_api.configuration.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

ledger_api.configuration.list (Testing)

Summary: Obtain the ledger configuration

Arguments:

- `expectedConfigs`: Int
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration

Return type:

- Seq[com.daml.ledger.api.v1.ledger_configuration_service.LedgerConfiguration]

Description: Returns the current ledger configuration and subsequent updates until the expected number of configs was retrieved or the timeout is over.

Ledger Api User Management Service

ledger_api.users.create (Testing)

Summary: Create a user with the given id

Arguments:

- `id`: String
- `actAs`: Set[com.digitalasset.canton.topology.PartyId]
- `primaryParty`: Option[com.digitalasset.canton.topology.PartyId]
- `readAs`: Set[com.digitalasset.canton.topology.PartyId]
- `participantAdmin`: Boolean
- `isActive`: Boolean
- `annotations`: Map[String,String]
- `identityProviderId`: String

Return type:

- com.digitalasset.canton.admin.api.client.data.User

Description: Users are used to dynamically managing the rights given to Daml applications. They allow us to link a stable local identifier (of an application) with a set of parties. id: the id used to identify the given user actAs: the set of parties this user is allowed to act as primaryParty: the optional party that should be linked to this user by default readAs: the set of parties this user is allowed to read as participantAdmin: flag (default false) indicating if the user is allowed to use the admin commands of the Ledger Api isActive: flag (default true) indicating if the user is active annotations: the set of key-value pairs linked to this user identityProviderId: identity provider id

ledger_api.users.delete (Testing)

Summary: Delete a user

Arguments:

- `id`: String
- `identityProviderId`: String

Description: Delete a user by id. id: user id identityProviderId: identity provider id

ledger_api.users.get (Testing)

Summary: Get the user data of the user with the given id

Arguments:

- `id`: String
- `identityProviderId`: String

Return type:

- com.digitalasset.canton.admin.api.client.data.User

Description: Fetch the data associated with the given user id failing if there is no such user. You will get the user's primary party, active status and annotations. If you need the user rights, use rights.list instead. id: user id identityProviderId: identity provider id

ledger_api.users.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

ledger_api.users.list (Testing)

Summary: List users

Arguments:

- `filterUser`: String
- `pageToken`: String
- `pageSize`: Int
- `identityProviderId`: String

Return type:

- com.digitalasset.canton.admin.api.client.data.UsersPage

Description: List users of this participant node filterUser: filter results using the given filter string pageToken: used for pagination (the result contains a page token if there are further pages) pageSize: default page size before the filter is applied identityProviderId: identity provider id

ledger_api.users.update (Testing)

Summary: Update a user

Arguments:

- `id`: String
- `modifier`: com.digitalasset.canton.admin.api.client.data.User => com.digitalasset.canton.admin.api.client.data.User
- `identityProviderId`: String

Return type:

- com.digitalasset.canton.admin.api.client.data.User

Description: Currently you can update the annotations, active status and primary party. You cannot update other user attributes. id: id of the user to be updated modifier: a function for modifying the user; e.g: *user => { user.copy(isActive = false, primaryParty = None, annotations = user.annotations.updated(a , b).removed(c))}* identityProviderId: identity provider id

ledger_api.users.update_idp (Testing)

Summary: Update user's identity provider id

Arguments:

- `id`: String
- `sourceIdentityProviderId`: String
- `targetIdentityProviderId`: String

Description: Updates user's identity provider id. id: the id used to identify the given user

sourceIdentityProviderId: source identity provider id targetIdentityProviderId: target identity provider id

ledger_api.users.rights.grant (Testing)

Summary: Grant new rights to a user

Arguments:

- `id`: String
- `actAs`: Set[com.digitalasset.canton.topology.PartyId]
- `readAs`: Set[com.digitalasset.canton.topology.PartyId]
- `participantAdmin`: Boolean
- `identityProviderId`: String

Return type:

- com.digitalasset.canton.admin.api.client.data.UserRights

Description: Users are used to dynamically managing the rights given to Daml applications. This function is used to grant new rights to an existing user. id: the id used to identify the given user actAs: the set of parties this user is allowed to act as readAs: the set of parties this user is allowed to read as participantAdmin: flag (default false) indicating if the user is allowed to use the admin commands of the Ledger Api identityProviderId: identity provider id

ledger_api.users.rights.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

ledger_api.users.rights.list (Testing)

Summary: List rights of a user

Arguments:

- `id`: String
- `identityProviderId`: String

Return type:

- com.digitalasset.canton.admin.api.client.data.UserRights

Description: Lists the rights of a user, or the rights of the current user. id: user id identityProviderId: identity provider id

ledger_api.users.rights.revoke (Testing)

Summary: Revoke user rights

Arguments:

- `id`: String
- `actAs`: Set[com.digitalasset.canton.topology.PartyId]
- `readAs`: Set[com.digitalasset.canton.topology.PartyId]
- `participantAdmin`: Boolean
- `identityProviderId`: String

Return type:

- com.digitalasset.canton.admin.api.client.data.UserRights

Description: Use to revoke specific rights from a user. id: the id used to identify the given user actAs: the set of parties this user should not be allowed to act as readAs: the set of parties this user should not be allowed to read as participantAdmin: if set to true, the participant admin rights will be removed identityProviderId: identity provider id

Ledger Api Metering Service

ledger_api.metering.get_report (Testing)

Summary: Get the ledger metering report
Arguments:
- `from`: com.digitalasset.canton.data.CantonTimestamp
- `to`: Option[com.digitalasset.canton.data.CantonTimestamp]
- `applicationId`: Option[String]

Return type:
- String

Description: Returns the current ledger metering report from: required from timestamp (inclusive) to: optional to timestamp application_id: optional application id to which we want to restrict the report

ledger_api.metering.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

Composability

transfer.execute (Preview)

Summary: Transfer the contract from the origin domain to the target domain
Arguments:
- `submittingParty`: com.digitalasset.canton.topology.PartyId
- `contractId`: com.digitalasset.canton.protocol.LfContractId
- `sourceDomain`: com.digitalasset.canton.DomainAlias
- `targetDomain`: com.digitalasset.canton.DomainAlias

Description: Macro that first calls transfer_out and then transfer_in. No error handling is done.

transfer.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

transfer.in (Preview)

Summary: Transfer-in a contract in transit to the target domain
Arguments:
- `submittingParty`: com.digitalasset.canton.topology.PartyId
- `transferId`: com.digitalasset.canton.protocol.TransferId
- `targetDomain`: com.digitalasset.canton.DomainAlias
- `applicationId`: com.digitalasset.canton.LedgerApplicationId
- `submissionId`: String
- `workflowId`: String
- `commandId`: String

Description: Manually transfers a contract in transit into the target domain. The command returns when the transfer-in has completed successfully. If the transferExclusivityTimeout in the target domain's parameters is set to a positive value, all participants of all

stakeholders connected to both origin and target domain will attempt to transfer-in the contract automatically after the exclusivity timeout has elapsed. An application-id can be specified to uniquely identifies the application that have issued the transfer, otherwise the default value will be used. An optional submission id can be set by the committer to the value of their choice that allows an application to correlate completions to its submissions.

transfer.lookup_contract_domain (Preview)

Summary: Lookup the active domain for the provided contracts

Arguments:

- `contractIds`: com.digitalasset.canton.protocol.LfContractId*

Return type:

- Map[com.digitalasset.canton.protocol.LfContractId,String]

transfer.out (Preview)

Summary: Transfer-out a contract from the source domain with destination target domain

Arguments:

- `submittingParty`: com.digitalasset.canton.topology.PartyId
- `contractId`: com.digitalasset.canton.protocol.LfContractId
- `sourceDomain`: com.digitalasset.canton.DomainAlias
- `targetDomain`: com.digitalasset.canton.DomainAlias
- `applicationId`: com.digitalasset.canton.LedgerApplicationId
- `submissionId`: String
- `workflowId`: String
- `commandId`: String

Return type:

- com.digitalasset.canton.protocol.TransferId

Description: Transfers the given contract out of the source domain with destination target domain. The command returns the ID of the transfer when the transfer-out has completed successfully. The contract is in transit until the transfer-in has completed on the target domain. The submitting party must be a stakeholder of the contract and the participant must have submission rights for the submitting party on the source domain. It must also be connected to the target domain. An application-id can be specified to uniquely identify the application that have issued the transfer, otherwise the default value will be used. An optional submission id can be set by the committer to the value of their choice that allows an application to correlate completions to its submissions.

transfer.search (Preview)

Summary: Search the currently in-flight transfers

Arguments:

- `targetDomain`: com.digitalasset.canton.DomainAlias
- `filterSourceDomain`: Option[com.digitalasset.canton.DomainAlias]
- `filterTimestamp`: Option[java.time.Instant]
- `filterSubmittingParty`: Option[com.digitalasset.canton.topology.PartyId]
- `limit`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:

- Seq[com.digitalasset.canton.participant.admin.grpc.TransferSearchResult]

Description: Returns all in-flight transfers with the given target domain that match the filters, but no more than the limit specifies.

Ledger Pruning

pruning.clear_schedule

Summary: Deactivate automatic pruning.

pruning.find_safe_offset (Preview)

Summary: Return the highest participant ledger offset whose record time is before or at the given one (if any) at which pruning is safely possible

Arguments:

- `beforeOrAt`: java.time.Instant

Return type:

- Option[com.daml.ledger.api.v1.ledger_offset.LedgerOffset]

pruning.get_offset_by_time

Summary: Identify the participant ledger offset to prune up to based on the specified timestamp.

Arguments:

- `upToInclusive`: java.time.Instant

Return type:

- Option[com.daml.ledger.api.v1.ledger_offset.LedgerOffset]

Description: Return the largest participant ledger offset that has been processed before or at the specified timestamp. The time is measured on the participant's local clock at some point while the participant has processed the the event. Returns `None` if no such offset exists.

pruning.get_schedule

Summary: Inspect the automatic pruning schedule.

Return type:

- Option[com.digitalasset.canton.admin.api.client.data.PruningSchedule]

Description: The schedule consists of a cron expression and max_duration and retention durations. The cron string indicates the points in time at which pruning should begin in the GMT time zone, and the maximum duration indicates how long from the start time pruning is allowed to run as long as pruning has not finished pruning up to the specified retention period. Returns *None* if no schedule has been configured via *set_schedule* or if *clear_schedule* has been invoked.

pruning.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

pruning.locate_offset (Preview)

Summary: Identify the participant ledger offset to prune up to.

Arguments:

- `n`: Long

Return type:

- com.daml.ledger.api.v1.ledger_offset.LedgerOffset

Description: Return the participant ledger offset that corresponds to pruning n number of transactions from the beginning of the ledger. Errors if the ledger holds less than n transactions. Specifying n of 1 returns the offset of the first transaction (if the ledger is non-empty).

pruning.prune

Summary: Prune the ledger up to the specified offset inclusively.
Arguments:
- `pruneUpTo`: com.daml.ledger.api.v1.ledger_offset.LedgerOffset

Description: Prunes the participant ledger up to the specified offset inclusively returning `Unit` if the ledger has been successfully pruned. Note that upon successful pruning, subsequent attempts to read transactions via `ledger_api.transactions.flat` or `ledger_api.transactions.trees` or command completions via `ledger_api.completions.list` by specifying a begin offset lower than the returned pruning offset will result in a `NOT_FOUND` error. In the Enterprise Edition, `prune` performs a full prune freeing up significantly more space and also performs additional safety checks returning a `NOT_FOUND` error if `pruneUpTo` is higher than the offset returned by `find_safe_offset` on any domain with events preceding the pruning offset.

pruning.prune_internally (Preview)

Summary: Prune only internal ledger state up to the specified offset inclusively.
Arguments:
- `pruneUpTo`: com.daml.ledger.api.v1.ledger_offset.LedgerOffset

Description: Special-purpose variant of the `prune` command only available in the Enterprise Edition that prunes only partial, internal participant ledger state freeing up space not needed for serving `ledger_api.transactions` and `ledger_api.completions` requests. In conjunction with `prune`, `prune_internally` enables pruning internal ledger state more aggressively than externally observable data via the ledger api. In most use cases `prune` should be used instead. Unlike `prune`, `prune_internally` has no visible effect on the Ledger API. The command returns `Unit` if the ledger has been successfully pruned or an error if the timestamp performs additional safety checks returning a `NOT_FOUND` error if `pruneUpTo` is higher than the offset returned by `find_safe_offset` on any domain with events preceding the pruning offset.

pruning.set_cron

Summary: Modify the cron used by automatic pruning.
Arguments:
- `cron`: String

Description: The schedule is specified in cron format and refers to pruning start times in the GMT time zone. This call returns an error if no schedule has been configured via *set_schedule* or if automatic pruning has been disabled via *clear_schedule*. Additionally if at the time of this modification, pruning is actively running, a best effort is made to pause pruning and restart according to the new schedule. This allows for the case that the new schedule no longer allows pruning at the current time.

pruning.set_max_duration

Summary: Modify the maximum duration used by automatic pruning.
Arguments:
- `maxDuration`: com.digitalasset.canton.config.PositiveDurationSeconds

Description: The *maxDuration* is specified as a positive duration and has at most per-second granularity. This call returns an error if no schedule has been configured via *set_schedule* or if automatic pruning has been disabled via *clear_schedule*. Additionally if at the time of this modification, pruning is actively running, a best effort is made to pause pruning and restart according to the new schedule. This allows for the case that the new schedule no longer allows pruning at the current time.

pruning.set_retention

Summary: Update the pruning retention used by automatic pruning.
Arguments:

- `retention`: com.digitalasset.canton.config.PositiveDurationSeconds

Description: The *retention* is specified as a positive duration and has at most per-second granularity. This call returns an error if no schedule has been configured via *set_schedule* or if automatic pruning has been disabled via *clear_schedule*. Additionally if at the time of this update, pruning is actively running, a best effort is made to pause pruning and restart with the newly specified retention. This allows for the case that the new retention mandates retaining more data than previously.

pruning.set_schedule

Summary: Activate automatic pruning according to the specified schedule.

Arguments:

- `cron`: String
- `maxDuration`: com.digitalasset.canton.config.PositiveDurationSeconds
- `retention`: com.digitalasset.canton.config.PositiveDurationSeconds

Description: The schedule is specified in cron format and max_duration and retention durations. The cron string indicates the points in time at which pruning should begin in the GMT time zone, and the maximum duration indicates how long from the start time pruning is allowed to run as long as pruning has not finished pruning up to the specified retention period.

Bilateral Commitments

commitments.computed

Summary: Lookup ACS commitments locally computed as part of the reconciliation protocol

Arguments:

- `domain`: com.digitalasset.canton.DomainAlias
- `start`: java.time.Instant
- `end`: java.time.Instant
- `counterParticipant`: Option[com.digitalasset.canton.topology.ParticipantId]

Return type:

- Iterable[(com.digitalasset.canton.protocol.messages.CommitmentPeriod, com.digitalasset.canton.topology.ParticipantId, com.digitalasset.canton.protocol.messages.AcsCommitment.CommitmentType)]

commitments.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

commitments.received

Summary: Lookup ACS commitments received from other participants as part of the reconciliation protocol

Arguments:

- `domain`: com.digitalasset.canton.DomainAlias
- `start`: java.time.Instant
- `end`: java.time.Instant
- `counterParticipant`: Option[com.digitalasset.canton.topology.ParticipantId]

Return type:

- Iterable[com.digitalasset.canton.protocol.messages.SignedProtocolMessage[com.digitalasset.canton.protocol.messages.AcsCommitment]]

Description: The arguments are: - domain: the alias of the domain - start: lowest time exclusive - end: highest time inclusive - counterParticipant: optionally filter by counter participant

Participant Repair

repair.add

Summary: Add specified contracts to specific domain on local participant.
Arguments:

- `domain`: com.digitalasset.canton.DomainAlias
- `contractsToAdd`: Seq[com.digitalasset.canton.protocol.SerializableContractWithWitnesses]
- `ignoreAlreadyAdded`: Boolean
- `ignoreStakeholderCheck`: Boolean

Description: This is a last resort command to recover from data corruption, e.g. in scenarios in which participant contracts have somehow gotten out of sync and need to be manually created. The participant needs to be disconnected from the specified domain at the time of the call, and as of now the domain cannot have had any inflight requests. For each contractsToAdd , specify witnesses , local parties, in case no local party is a stakeholder. The ignoreAlreadyAdded flag makes it possible to invoke the command multiple times with the same parameters in case an earlier command invocation has failed. As repair commands are powerful tools to recover from unforeseen data corruption, but dangerous under normal operation, use of this command requires (temporarily) enabling the features.enable-repair-commands configuration. In addition repair commands can run for an unbounded time depending on the number of contracts passed in. Be sure to not connect the participant to the domain until the call returns. The arguments are: - domain: the alias of the domain to which to add the contract - contractsToAdd: list of contracts to add with witness information - ignoreAlreadyAdded: (default true) if set to true, it will ignore contracts that already exist on the target domain. - ignoreStakeholderCheck: (default false) if set to true, add will work for contracts that don't have a local party (useful for party migration).

repair.change_domain

Summary: Move contracts with specified Contract IDs from one domain to another.
Arguments:

- `contractIds`: Seq[com.digitalasset.canton.protocol.LfContractId]
- `sourceDomain`: com.digitalasset.canton.DomainAlias
- `targetDomain`: com.digitalasset.canton.DomainAlias
- `skipInactive`: Boolean
- `batchSize`: Int

Description: This is a last resort command to recover from data corruption in scenarios in which a domain is irreparably broken and formerly connected participants need to move contracts to another, healthy domain. The participant needs to be disconnected from both the sourceDomain and the targetDomain . Also as of now the target domain cannot have had any inflight requests. Contracts already present in the target domain will be skipped, and this makes it possible to invoke this command in an idempotent fashion in case an earlier attempt had resulted in an error. The skipInactive flag makes it possible to only move active contracts in the sourceDomain . As repair commands are powerful tools to recover from unforeseen data corruption, but dangerous under normal operation, use of this command requires (temporarily) enabling the features.enable-repair-commands configuration. In addition repair commands can run for

an unbounded time depending on the number of contract ids passed in. Be sure to not connect the participant to either domain until the call returns. Arguments: - contractIds - set of contract ids that should be moved to the new domain - sourceDomain - alias of the source domain - targetDomain - alias of the target domain - skipInactive - (default true) whether to skip inactive contracts mentioned in the contractIds list - batchSize - (default 100) how many contracts to write at once to the database

repair.download

Summary: Download all contracts for the given set of parties to a file.

Arguments:

- `parties`: Set[com.digitalasset.canton.topology.PartyId]
- `outputFile`: String
- `filterDomainId`: String
- `timestamp`: Option[java.time.Instant]
- `protocolVersion`: Option[com.digitalasset.canton.version.ProtocolVersion]
- `chunkSize`: Option[com.digitalasset.canton.config.RequireTypes.PositiveInt]
- `contractDomainRenames`: Map[com.digitalasset.canton.topology.DomainId,com.digitalasset.canton.topology.DomainId]

Description: This command can be used to download the current active contract set of a given set of parties to a text file. This is mainly interesting for recovery and operational purposes. The file will contain base64 encoded strings, one line per contract. The lines are written sorted according to their domain and contract id. This allows to compare the contracts stored by two participants using standard file comparison tools. The domain-id is printed with the prefix domain-id before the block of contracts starts. This command may take a long time to complete and may require significant resources. It will first load the contract ids of the active contract set into memory and then subsequently load the contracts in batches and inspect their stakeholders. As this operation needs to traverse the entire datastore, it might take a long time to complete. The command will return a map of domainId -> number of active contracts stored The arguments are: - parties: identifying contracts having at least one stakeholder from the given set - outputFile: the output file name where to store the data. Use .gz as a suffix to get a compressed file (recommended) - filterDomainId: restrict the export to a given domain - timestamp: optionally a timestamp for which we should take the state (useful to reconcile states of a domain) - protocolVersion: optional the protocol version to use for the serialization. Defaults to the one of the domains. - chunkSize: size of the byte chunks to stream back: default 1024 * 1024 * 2 = (2MB) - contractDomainRenames: As part of the export, allow to rename the associated domain id of contracts from one domain to another based on the mapping.

repair.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

repair.ignore_events

Summary: Mark sequenced events as ignored.

Arguments:

- `domainId`: com.digitalasset.canton.topology.DomainId
- `from`: com.digitalasset.canton.SequencerCounter
- `to`: com.digitalasset.canton.SequencerCounter
- `force`: Boolean

Description: This is the last resort to ignore events that the participant is unable to process. Ignoring events may lead to subsequent failures, e.g., if the event creating a contract

is ignored and that contract is subsequently used. It may also lead to ledger forks if other participants still process the ignored events. It is possible to mark events as ignored that the participant has not yet received. The command will fail, if marking events between *from* and *to* as ignored would result in a gap in sequencer counters, namely if *from <= to* and *from* is greater than *maxSequencerCounter + 1*, where *maxSequencerCounter* is the greatest sequencer counter of a sequenced event stored by the underlying participant. The command will also fail, if *force == false* and *from* is smaller than the sequencer counter of the last event that has been marked as clean. (Ignoring such events would normally have no effect, as they have already been processed.)

repair.migrate_domain

Summary: Migrate contracts from one domain to another one.

Arguments:

- `source`: com.digitalasset.canton.DomainAlias
- `target`: com.digitalasset.canton.participant.domain.DomainConnectionConfig

Description: This method can be used to migrate all the contracts associated with a domain to a new domain connection. This method will register the new domain, connect to it and then re-associate all contracts on the source domain to the target domain. Please note that this migration needs to be done by all participants at the same time. The domain should only be used once all participants have finished their migration. The arguments are: source: the domain alias of the source domain target: the configuration for the target domain

repair.purge

Summary: Purge contracts with specified Contract IDs from local participant.

Arguments:

- `domain`: com.digitalasset.canton.DomainAlias
- `contractIds`: Seq[com.digitalasset.canton.protocol.LfContractId]
- `ignoreAlreadyPurged`: Boolean

Description: This is a last resort command to recover from data corruption, e.g. in scenarios in which participant contracts have somehow gotten out of sync and need to be manually purged, or in situations in which stakeholders are no longer available to agree to their archival. The participant needs to be disconnected from the domain on which the contracts with contractIds reside at the time of the call, and as of now the domain cannot have had any inflight requests. The ignoreAlreadyPurged flag makes it possible to invoke the command multiple times with the same parameters in case an earlier command invocation has failed. As repair commands are powerful tools to recover from unforeseen data corruption, but dangerous under normal operation, use of this command requires (temporarily) enabling the features.enable-repair-commands configuration. In addition repair commands can run for an unbounded time depending on the number of contract ids passed in. Be sure to not connect the participant to the domain until the call returns.

repair.unignore_events

Summary: Remove the ignored status from sequenced events.

Arguments:

- `domainId`: com.digitalasset.canton.topology.DomainId
- `from`: com.digitalasset.canton.SequencerCounter
- `to`: com.digitalasset.canton.SequencerCounter
- `force`: Boolean

Description: This command has no effect on ordinary (i.e., not ignored) events and on events that do not exist. The command will fail, if marking events between *from* and *to* as unignored would result in a gap in sequencer counters, namely if there is one empty

ignored event with sequencer counter between *from* and *to* and another empty ignored event with sequencer counter greater than *to*. An empty ignored event is an event that has been marked as ignored and not yet received by the participant. The command will also fail, if *force == false* and *from* is smaller than the sequencer counter of the last event that has been marked as clean. (Unignoring such events would normally have no effect, as they have already been processed.)

repair.upload

Summary: Import ACS snapshot
Arguments:
- `inputFile`: String

Description: Uploads a binary into the participant's ACS

Resource Management

resources.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

resources.resource_limits

Summary: Get the resource limits of the participant.
Return type:
- com.digitalasset.canton.participant.admin.ResourceLimits

resources.set_resource_limits

Summary: Set resource limits for the participant.
Arguments:
- `limits`: com.digitalasset.canton.participant.admin.ResourceLimits

Description: While a resource limit is attained or exceeded, the participant will reject any additional submission with GRPC status ABORTED. Most importantly, a submission will be rejected **before** it consumes a significant amount of resources. There are three kinds of limits: *maxDirtyRequests*, *maxRate* and *maxBurstFactor*. The number of dirty requests of a participant P covers (1) requests initiated by P as well as (2) requests initiated by participants other than P that need to be validated by P. Compared to the maximum rate, the maximum number of dirty requests reflects the load on the participant more accurately. However, the maximum number of dirty requests alone does not protect the system from bursts : If an application submits a huge number of commands at once, the maximum number of dirty requests will likely be exceeded, as the system is registering dirty requests only during validation and not already during submission. The maximum rate is a hard limit on the rate of commands submitted to this participant through the ledger API. As the rate of commands is checked and updated immediately after receiving a new command submission, an application cannot exceed the maximum rate. The *maxBurstFactor* parameter (positive, default 0.5) allows to configure how permissive the rate limitation should be with respect to bursts. The rate limiting will be enforced strictly after having observed *max_burst * max_rate* commands. For the sake of illustration, let's assume the configured rate limit is `100 commands/s` with a burst ratio of 0.5. If an application submits 100 commands within a single second, waiting exactly 10 milliseconds between consecutive commands, then the participant will accept all commands. With a *maxBurstFactor* of 0.5, the participant will accept the first 50 commands and reject the remaining 50. If the application then waits another 500 ms, it may submit another burst of 50 commands. If it

waits 250 ms, it may submit only a burst of 25 commands. Resource limits can only be changed, if the server runs Canton enterprise. In the community edition, the server uses fixed limits that cannot be changed.

Replication

replication.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

replication.set_passive

Summary: Set the participant replica to passive

Description: Trigger a graceful fail-over from this active replica to another passive replica.

1.28.3.3 Multiple Participants

This section lists the commands available for a sequence of participants. They can be used on the participant references `participants.all`, `.local` or `.remote` as:

```
participants.all.dars.upload("my.dar")
```

dars.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

dars.upload

Summary: Upload DARs to participants

Arguments:

- `darPath`: String
- `vetAllPackages`: Boolean
- `synchronizeVetting`: Boolean

Return type:

- Map[com.digitalasset.canton.console.ParticipantReference,String]

Description: If vetAllPackages is true, the participants will vet the package on all domains they are registered. If synchronizeVetting is true, the command will block until the package vetting transaction has been registered with all connected domains.

domains.connect_local

Summary: Register and potentially connect to new local domain

Arguments:

- `domain`: com.digitalasset.canton.console.InstanceReferenceWithSequencerConnection
- `manualConnect`: Boolean
- `synchronize`: Option[com.digitalasset.canton.config.NonNegativeDuration]

Description: The arguments are: domain - A local domain or sequencer reference manualConnect - Whether this connection should be handled manually and also excluded from automatic re-connect. synchronize - A timeout duration indicating how long to wait for all topology changes to have been effected on all local nodes.

domains.disconnect
Summary: Disconnect from domain
Arguments:
- `alias`: com.digitalasset.canton.DomainAlias

domains.disconnect_all
Summary: Disconnect from all connected domains

domains.disconnect_local
Summary: Disconnect from a local domain
Arguments:
- `domain`: com.digitalasset.canton.console.LocalDomainReference

domains.help
Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

domains.reconnect
Summary: Reconnect to domain
Arguments:
- `alias`: com.digitalasset.canton.DomainAlias
- `retry`: Boolean

Description: If retry is set to true (default), the command will return after the first attempt, but keep on trying in the background.

domains.reconnect_all
Summary: Reconnect to all domains for which *manualStart* = false
Arguments:
- `ignoreFailures`: Boolean

Description: If ignoreFailures is set to true (default), the reconnect all will succeed even if some domains are offline. The participants will continue attempting to establish a domain connection.

domains.register
Summary: Register and potentially connect to domain
Arguments:
- `config`: com.digitalasset.canton.participant.domain.DomainConnectionConfig

1.28.3.4 Domain Administration Commands

clear_cache (Testing)
Summary: Clear locally cached variables
Description: Some commands cache values on the client side. Use this command to explicitly clear the caches of these values.

config
Summary: Returns the domain configuration
Return type:
- LocalDomainReference.this.consoleEnvironment.environment.config.DomainConfigType

defaultDomainConnection

Summary: Yields a domain connection config with default values except for the domain alias and the sequencer connection. May throw an exception if the domain alias or sequencer connection is misconfigured.
Return type:
- com.digitalasset.canton.participant.domain.DomainConnectionConfig

help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

id

Summary: Yields the globally unique id of this domain. Throws an exception, if the id has not yet been allocated (e.g., the domain has not yet been started).
Return type:
- com.digitalasset.canton.topology.DomainId

is_initialized

Summary: Check if the local instance is running and is fully initialized
Return type:
- Boolean

is_running

Summary: Check if the local instance is running
Return type:
- Boolean

start

Summary: Start the instance

stop

Summary: Stop the instance

Health

health.active

Summary: Check if the node is running and is the active instance (mediator, participant)
Return type:
- Boolean

health.dump

Summary: Creates a zip file containing diagnostic information about the canton process running this node
Arguments:
- `outputFile`: better.files.File
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration
- `chunkSize`: Option[Int]

Return type:
- String

health.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:
- `methodName`: String

health.initialized

Summary: Returns true if node has been initialized.
Return type:
- Boolean

health.running

Summary: Check if the node is running
Return type:
- Boolean

health.status

Summary: Get human (and machine) readable status info
Return type:
- com.digitalasset.canton.health.admin.data.NodeStatus[S]

health.wait_for_identity

Summary: Wait for the node to have an identity
Description: This is specifically useful for the Domain Manager which needs its identity to be ready for bootstrapping, but for which we can't rely on wait_for_initialized() because it will be initialized only after being bootstrapped.

health.wait_for_initialized

Summary: Wait for the node to be initialized

health.wait_for_running

Summary: Wait for the node to be running

Database

db.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

db.migrate

Summary: Migrates the instance's database if using a database storage

db.repair_migration

Summary: Only use when advised - repairs the database migration of the instance's database
Arguments:
- `force`: Boolean

Description: In some rare cases, we change already applied database migration files in a new release and the repair command resets the checksums we use to ensure that in general already applied migration files have not been changed. You should only use *db.repair_migration* when advised and otherwise use it at your own risk - in the worst case running it may lead to data corruption when an incompatible database migration (one that should be rejected because the already applied database migration files have changed) is subsequently falsely applied.

Participants

participants.active

Summary: Test whether a participant is permissioned on this domain
Arguments:
- `participantId`: com.digitalasset.canton.topology.ParticipantId

Return type:
- Boolean

participants.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

participants.list

Summary: List participant states
Return type:
- Seq[com.digitalasset.canton.admin.api.client.data.ListParticipantDomainStateResult]

Description: This command will list the currently valid state as stored in the authorized store. For a deep inspection of the identity management history, use the *topology.participant_domain_states.list* command.

participants.set_state

Summary: Change state and trust level of participant
Arguments:
- `participant`: com.digitalasset.canton.topology.ParticipantId
- `permission`: com.digitalasset.canton.topology.transaction.ParticipantPermission
- `trustLevel`: com.digitalasset.canton.topology.transaction.TrustLevel
- `synchronize`: Option[com.digitalasset.canton.config.NonNegativeDuration]

Description: Set the state of the participant within the domain. Valid permissions are 'Submission', 'Confirmation', 'Observation' and 'Disabled'. Valid trust levels are 'Vip' and 'Ordinary'. Synchronize timeout can be used to ensure that the state has been propagated into the node

Sequencer

sequencer.disable_member

Summary: Disable the provided member at the Sequencer that will allow any unread data for them to be removed
Arguments:
- `member`: com.digitalasset.canton.topology.Member

Description: This will prevent any client for the given member to reconnect the Sequencer and allow any unread/unacknowledged data they have to be removed. This should only be used if the domain operation is confident the member will never need to reconnect as there is no way to re-enable the member. To view members using the sequencer run *sequencer.status()*.

sequencer.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

sequencer.pruning.clear_schedule
Summary: Deactivate automatic pruning.

sequencer.pruning.force_prune
Summary: Force remove data from the Sequencer including data that may have not been read by offline clients
Arguments:
- `dryRun`: Boolean

Return type:
- String

Description: Will force pruning up until the default retention period by potentially disabling clients that have not yet read data we would like to remove. Disabling these clients will prevent them from ever reconnecting to the Domain so should only be used if the Domain operator is confident they can be permanently ignored. Run with *dryRun = true* to review a description of which clients will be disabled first. Run with *dryRun = false* to disable these clients and perform a forced pruning.

sequencer.pruning.force_prune_at
Summary: Force removing data from the Sequencer including data that may have not been read by offline clients up until the specified time
Arguments:
- `timestamp`: com.digitalasset.canton.data.CantonTimestamp
- `dryRun`: Boolean

Return type:
- String

Description: Similar to the above *force_prune* command but allows specifying the exact time at which to prune

sequencer.pruning.force_prune_with_retention_period
Summary: Force removing data from the Sequencer including data that may have not been read by offline clients up until a custom retention period
Arguments:
- `retentionPeriod`: scala.concurrent.duration.FiniteDuration
- `dryRun`: Boolean

Return type:
- String

Description: Similar to the above *force_prune* command but allows specifying a custom retention period

sequencer.pruning.get_schedule
Summary: Inspect the automatic pruning schedule.
Return type:
- Option[com.digitalasset.canton.admin.api.client.data.PruningSchedule]

Description: The schedule consists of a cron expression and max_duration and retention durations. The cron string indicates the points in time at which pruning should begin in the GMT time zone, and the maximum duration indicates how long from the start time pruning is allowed to run as long as pruning has not finished pruning up to the specified retention period. Returns *None* if no schedule has been configured via *set_schedule* or if *clear_schedule* has been invoked.

sequencer.pruning.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

sequencer.pruning.locate_pruning_timestamp

Summary: Obtain a timestamp at or near the beginning of sequencer state

Arguments:

- `index`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:

- Option[com.digitalasset.canton.data.CantonTimestamp]

Description: This command provides insight into the current state of sequencer pruning when called with the default value of *index* 1. When pruning the sequencer manually via *prune_at* and with the intent to prune in batches, specify a value such as 1000 to obtain a pruning timestamp that corresponds to the end of the batch.

sequencer.pruning.prune

Summary: Remove unnecessary data from the Sequencer up until the default retention point

Return type:

- String

Description: Removes unnecessary data from the Sequencer that is earlier than the default retention period. The default retention period is set in the configuration of the canton processing running this command under *parameters.retention-period-defaults.sequencer*. This pruning command requires that data is read and acknowledged by clients before considering it safe to remove. If no data is being removed it could indicate that clients are not reading or acknowledging data in a timely fashion (typically due to nodes going offline for long periods). You have the option of disabling the members running on these nodes to allow removal of this data, however this will mean that they will be unable to reconnect to the domain in the future. To do this run *force_prune(dryRun = true)* to return a description of which members would be disabled in order to prune the Sequencer. If you are happy to disable the described clients then run *force_prune(dryRun = false)* to permanently remove their unread data. Once offline clients have been disabled you can continue to run *prune* normally.

sequencer.pruning.prune_at

Summary: Remove data that has been read up until the specified time

Arguments:

- `timestamp`: com.digitalasset.canton.data.CantonTimestamp

Return type:

- String

Description: Similar to the above *prune* command but allows specifying the exact time at which to prune. The command will fail if a client has not yet read and acknowledged some data up to the specified time.

sequencer.pruning.prune_with_retention_period

Summary: Remove data that has been read up until a custom retention period

Arguments:

- `retentionPeriod`: scala.concurrent.duration.FiniteDuration

Return type:

- String

Description: Similar to the above *prune* command but allows specifying a custom reten-

tion period

sequencer.pruning.set_cron

Summary: Modify the cron used by automatic pruning.

Arguments:

- `cron`: String

Description: The schedule is specified in cron format and refers to pruning start times in the GMT time zone. This call returns an error if no schedule has been configured via *set_schedule* or if automatic pruning has been disabled via *clear_schedule*. Additionally if at the time of this modification, pruning is actively running, a best effort is made to pause pruning and restart according to the new schedule. This allows for the case that the new schedule no longer allows pruning at the current time.

sequencer.pruning.set_max_duration

Summary: Modify the maximum duration used by automatic pruning.

Arguments:

- `maxDuration`: com.digitalasset.canton.config.PositiveDurationSeconds

Description: The *maxDuration* is specified as a positive duration and has at most per-second granularity. This call returns an error if no schedule has been configured via *set_schedule* or if automatic pruning has been disabled via *clear_schedule*. Additionally if at the time of this modification, pruning is actively running, a best effort is made to pause pruning and restart according to the new schedule. This allows for the case that the new schedule no longer allows pruning at the current time.

sequencer.pruning.set_retention

Summary: Update the pruning retention used by automatic pruning.

Arguments:

- `retention`: com.digitalasset.canton.config.PositiveDurationSeconds

Description: The *retention* is specified as a positive duration and has at most per-second granularity. This call returns an error if no schedule has been configured via *set_schedule* or if automatic pruning has been disabled via *clear_schedule*. Additionally if at the time of this update, pruning is actively running, a best effort is made to pause pruning and restart with the newly specified retention. This allows for the case that the new retention mandates retaining more data than previously.

sequencer.pruning.set_schedule

Summary: Activate automatic pruning according to the specified schedule.

Arguments:

- `cron`: String
- `maxDuration`: com.digitalasset.canton.config.PositiveDurationSeconds
- `retention`: com.digitalasset.canton.config.PositiveDurationSeconds

Description: The schedule is specified in cron format and max_duration and retention durations. The cron string indicates the points in time at which pruning should begin in the GMT time zone, and the maximum duration indicates how long from the start time pruning is allowed to run as long as pruning has not finished pruning up to the specified retention period.

sequencer.pruning.status

Summary: Status of the sequencer and its connected clients

Return type:

- com.digitalasset.canton.domain.sequencing.sequencer.SequencerPruningStatus

Description: Provides a detailed breakdown of information required for pruning: - the current time according to this sequencer instance - domain members that the sequencer supports - for each member when they were registered and whether they are enabled - a

list of clients for each member, their last acknowledgement, and whether they are enabled

Mediator

mediator.clear_schedule

Summary: Deactivate automatic pruning.

mediator.get_schedule

Summary: Inspect the automatic pruning schedule.

Return type:

- Option[com.digitalasset.canton.admin.api.client.data.PruningSchedule]

Description: The schedule consists of a cron expression and max_duration and retention durations. The cron string indicates the points in time at which pruning should begin in the GMT time zone, and the maximum duration indicates how long from the start time pruning is allowed to run as long as pruning has not finished pruning up to the specified retention period. Returns *None* if no schedule has been configured via *set_schedule* or if *clear_schedule* has been invoked.

mediator.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

mediator.locate_pruning_timestamp

Summary: Obtain a timestamp at or near the beginning of mediator state

Arguments:

- `index`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:

- Option[com.digitalasset.canton.data.CantonTimestamp]

Description: This command provides insight into the current state of mediator pruning when called with the default value of *index* 1. When pruning the mediator manually via *prune_at* and with the intent to prune in batches, specify a value such as 1000 to obtain a pruning timestamp that corresponds to the end of the batch.

mediator.prune

Summary: Prune the mediator of unnecessary data while keeping data for the default retention period

Description: Removes unnecessary data from the Mediator that is earlier than the default retention period. The default retention period is set in the configuration of the canton node running this command under *parameters.retention-period-defaults.mediator*.

mediator.prune_at

Summary: Prune the mediator of unnecessary data up to and including the given timestamp

Arguments:

- `timestamp`: com.digitalasset.canton.data.CantonTimestamp

mediator.prune_with_retention_period

Summary: Prune the mediator of unnecessary data while keeping data for the provided retention period

Arguments:

- `retentionPeriod`: scala.concurrent.duration.FiniteDuration

mediator.set_cron

Summary: Modify the cron used by automatic pruning.
Arguments:
- `cron`: String

Description: The schedule is specified in cron format and refers to pruning start times in the GMT time zone. This call returns an error if no schedule has been configured via *set_schedule* or if automatic pruning has been disabled via *clear_schedule*. Additionally if at the time of this modification, pruning is actively running, a best effort is made to pause pruning and restart according to the new schedule. This allows for the case that the new schedule no longer allows pruning at the current time.

mediator.set_max_duration

Summary: Modify the maximum duration used by automatic pruning.
Arguments:
- `maxDuration`: com.digitalasset.canton.config.PositiveDurationSeconds

Description: The *maxDuration* is specified as a positive duration and has at most per-second granularity. This call returns an error if no schedule has been configured via *set_schedule* or if automatic pruning has been disabled via *clear_schedule*. Additionally if at the time of this modification, pruning is actively running, a best effort is made to pause pruning and restart according to the new schedule. This allows for the case that the new schedule no longer allows pruning at the current time.

mediator.set_retention

Summary: Update the pruning retention used by automatic pruning.
Arguments:
- `retention`: com.digitalasset.canton.config.PositiveDurationSeconds

Description: The *retention* is specified as a positive duration and has at most per-second granularity. This call returns an error if no schedule has been configured via *set_schedule* or if automatic pruning has been disabled via *clear_schedule*. Additionally if at the time of this update, pruning is actively running, a best effort is made to pause pruning and restart with the newly specified retention. This allows for the case that the new retention mandates retaining more data than previously.

mediator.set_schedule

Summary: Activate automatic pruning according to the specified schedule.
Arguments:
- `cron`: String
- `maxDuration`: com.digitalasset.canton.config.PositiveDurationSeconds
- `retention`: com.digitalasset.canton.config.PositiveDurationSeconds

Description: The schedule is specified in cron format and max_duration and retention durations. The cron string indicates the points in time at which pruning should begin in the GMT time zone, and the maximum duration indicates how long from the start time pruning is allowed to run as long as pruning has not finished pruning up to the specified retention period.

mediator.testing.await_domain_time (Testing)

Summary: Await for the given time to be reached on the domain
Arguments:
- `time`: com.digitalasset.canton.data.CantonTimestamp
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration

mediator.testing.fetch_domain_time (Testing)

Summary: Fetch the current time from the domain
Arguments:

- `timeout`: com.digitalasset.canton.config.NonNegativeDuration

Return type:

- com.digitalasset.canton.data.CantonTimestamp

mediator.testing.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

Key Administration

keys.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

keys.public.download

Summary: Download public key

Arguments:

- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint
- `protocolVersion`: com.digitalasset.canton.version.ProtocolVersion

Return type:

- com.google.protobuf.ByteString

keys.public.download_to

Summary: Download public key and save it to a file

Arguments:

- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint
- `outputFile`: String
- `protocolVersion`: com.digitalasset.canton.version.ProtocolVersion

keys.public.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

keys.public.list

Summary: List public keys in registry

Arguments:

- `filterFingerprint`: String
- `filterContext`: String

Return type:

- Seq[com.digitalasset.canton.crypto.PublicKeyWithName]

Description: Returns all public keys that have been added to the key registry. Optional arguments can be used for filtering.

keys.public.list_by_owner

Summary: List keys for given keyOwner.

Arguments:

- `keyOwner`: com.digitalasset.canton.topology.KeyOwner

- `filterDomain`: String
- `asOf`: Option[java.time.Instant]
- `limit`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:

- Seq[com.digitalasset.canton.admin.api.client.data.ListKeyOwnersResult]

Description: This command is a convenience wrapper for *list_key_owners*, taking an explicit keyOwner as search argument. The response includes the public keys.

keys.public.list_owners

Summary: List active owners with keys for given search arguments.

Arguments:

- `filterKeyOwnerUid`: String
- `filterKeyOwnerType`: Option[com.digitalasset.canton.topology.KeyOwnerCode]
- `filterDomain`: String
- `asOf`: Option[java.time.Instant]
- `limit`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:

- Seq[com.digitalasset.canton.admin.api.client.data.ListKeyOwnersResult]

Description: This command allows deep inspection of the topology state. The response includes the public keys. Optional filterKeyOwnerType type can be 'ParticipantId.Code' , 'MediatorId.Code','SequencerId.Code', 'DomainTopologyManagerId.Code'.

keys.public.upload

Summary: Upload public key

Arguments:

- `filename`: String
- `name`: Option[String]

Return type:

- com.digitalasset.canton.crypto.Fingerprint

keys.public.upload

Summary: Upload public key

Arguments:

- `keyBytes`: com.google.protobuf.ByteString
- `name`: Option[String]

Return type:

- com.digitalasset.canton.crypto.Fingerprint

Description: Import a public key and store it together with a name used to provide some context to that key.

keys.secret.delete

Summary: Delete private key

Arguments:

- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint
- `force`: Boolean

keys.secret.download

Summary: Download key pair

Arguments:

- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint
- `protocolVersion`: com.digitalasset.canton.version.ProtocolVersion

Return type:

- com.google.protobuf.ByteString

keys.secret.download_to

Summary: Download key pair and save it to a file
Arguments:
- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint
- `outputFile`: String
- `protocolVersion`: com.digitalasset.canton.version.ProtocolVersion

keys.secret.generate_encryption_key

Summary: Generate new public/private key pair for encryption and store it in the vault
Arguments:
- `name`: String
- `scheme`: Option[com.digitalasset.canton.crypto.EncryptionKeyScheme]

Return type:
- com.digitalasset.canton.crypto.EncryptionPublicKey

Description: The optional name argument allows you to store an associated string for your convenience. The scheme can be used to select a key scheme and the default scheme is used if left unspecified.

keys.secret.generate_signing_key

Summary: Generate new public/private key pair for signing and store it in the vault
Arguments:
- `name`: String
- `scheme`: Option[com.digitalasset.canton.crypto.SigningKeyScheme]

Return type:
- com.digitalasset.canton.crypto.SigningPublicKey

Description: The optional name argument allows you to store an associated string for your convenience. The scheme can be used to select a key scheme and the default scheme is used if left unspecified.

keys.secret.get_wrapper_key_id

Summary: Get the wrapper key id that is used for the encrypted private keys store
Return type:
- String

keys.secret.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

keys.secret.list

Summary: List keys in private vault
Arguments:
- `filterFingerprint`: String
- `filterName`: String
- `purpose`: Set[com.digitalasset.canton.crypto.KeyPurpose]

Return type:
- Seq[com.digitalasset.canton.crypto.admin.grpc.PrivateKeyMetadata]

Description: Returns all public keys to the corresponding private keys in the key vault. Optional arguments can be used for filtering.

keys.secret.register_kms_encryption_key

Summary: Register the specified KMS encryption key in canton storing its public information in the vault

Arguments:
- `kmsKeyId`: String
- `name`: String

Return type:
- com.digitalasset.canton.crypto.EncryptionPublicKey

Description: The id for the KMS encryption key. The optional name argument allows you to store an associated string for your convenience.

keys.secret.register_kms_signing_key

Summary: Register the specified KMS signing key in canton storing its public information in the vault

Arguments:
- `kmsKeyId`: String
- `name`: String

Return type:
- com.digitalasset.canton.crypto.SigningPublicKey

Description: The id for the KMS signing key. The optional name argument allows you to store an associated string for your convenience.

keys.secret.rotate_kms_node_key

Summary: Rotate a given node's keypair with a new pre-generated KMS keypair

Arguments:
- `fingerprint`: String
- `newKmsKeyId`: String

Return type:
- com.digitalasset.canton.crypto.PublicKey

Description: Rotates an existing encryption or signing key stored externally in a KMS with a pre-generated key. The fingerprint of the key we want to rotate. The id of the new KMS key (e.g. Resource Name).

keys.secret.rotate_node_key

Summary: Rotate a node's public/private key pair

Arguments:
- `fingerprint`: String
- `name`: Option[String]

Return type:
- com.digitalasset.canton.crypto.PublicKey

Description: Rotates an existing encryption or signing key. NOTE: A namespace root or intermediate signing key CANNOT be rotated by this command. The fingerprint of the key we want to rotate.

keys.secret.rotate_node_keys

Summary: Rotate the node's public/private key pairs

Description: For a participant node it rotates the signing and encryption key pair. For a domain or domain manager node it rotates the signing key pair as those nodes do not have an encryption key pair. For a sequencer or mediator node use *rotate_node_keys* with a domain manager reference as an argument. NOTE: Namespace root or intermediate signing keys are NOT rotated by this command.

keys.secret.rotate_wrapper_key

Summary: Change the wrapper key for encrypted private keys store

Arguments:
- `newWrapperKeyId`: String

Description: Change the wrapper key (e.g. AWS KMS key) being used to encrypt the private

keys in the store. newWrapperKeyId: The optional new wrapper key id to be used. If the wrapper key id is empty Canton will generate a new key based on the current configuration.

keys.secret.upload

Summary: Upload a key pair

Arguments:

- `pairBytes`: com.google.protobuf.ByteString
- `name`: Option[String]

keys.secret.upload

Summary: Upload (load and import) a key pair from file

Arguments:

- `filename`: String
- `name`: Option[String]

Parties

parties.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

parties.list

Summary: List active parties, their active participants, and the participants' permissions on domains.

Arguments:

- `filterParty`: String
- `filterParticipant`: String
- `filterDomain`: String
- `asOf`: Option[java.time.Instant]
- `limit`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:

- Seq[com.digitalasset.canton.admin.api.client.data.ListPartiesResult]

Description: Inspect the parties known by this participant as used for synchronisation. The response is built from the timestamped topology transactions of each domain, excluding the authorized store of the given node. For each known party, the list of active participants and their permission on the domain for that party is given. filterParty: Filter by parties starting with the given string. filterParticipant: Filter for parties that are hosted by a participant with an id starting with the given string filterDomain: Filter by domains whose id starts with the given string. asOf: Optional timestamp to inspect the topology state at a given point in time. limit: Limit on the number of parties fetched (defaults to canton.parameters.console.default-limit). Example: participant1.parties.list(filterParty= alice)

Service

service.get_dynamic_domain_parameters
Summary: Get the Dynamic Domain Parameters configured for the domain
Return type:
- com.digitalasset.canton.admin.api.client.data.DynamicDomainParameters

service.get_max_rate_per_participant
Summary: Get the max rate per participant
Return type:
- com.digitalasset.canton.config.RequireTypes.NonNegativeInt

Description: Depending on the protocol version used on the domain, the value will be read either from the static domain parameters or the dynamic ones.

service.get_max_request_size
Summary: Get the max request size
Return type:
- com.digitalasset.canton.config.RequireTypes.NonNegativeInt

Description: Depending on the protocol version used on the domain, the value will be read either from the static domain parameters or the dynamic ones. This value is not necessarily the one used by the sequencer node because it requires a restart of the server to be taken into account.

service.get_mediator_deduplication_timeout
Summary: Get the mediator deduplication timeout
Return type:
- com.digitalasset.canton.config.NonNegativeFiniteDuration

Description: The method will fail, if the domain does not support the mediatorDeduplicationTimeout.

service.get_reconciliation_interval
Summary: Get the reconciliation interval configured for the domain
Return type:
- com.digitalasset.canton.config.PositiveDurationSeconds

Description: Depending on the protocol version used on the domain, the value will be read either from the static domain parameters or the dynamic ones.

service.get_static_domain_parameters
Summary: Get the Static Domain Parameters configured for the domain
Return type:
- com.digitalasset.canton.admin.api.client.data.StaticDomainParameters

service.help
Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

service.list_accepted_agreements
Summary: List the accepted service agreements
Return type:
- Seq[com.digitalasset.canton.domain.service.ServiceAgreementAcceptance]

service.set_dynamic_domain_parameters
Summary: Set the Dynamic Domain Parameters configured for the domain

Arguments:

- `dynamicDomainParameters`: com.digitalasset.canton.admin.api.client.data.DynamicDomainParameters
- `force`: Boolean

Description: force: Enable potentially dangerous changes. Required to increase `ledgerTimeRecordTimeTolerance`. Use `set_ledger_time_record_time_tolerance` to securely increase `ledgerTimeRecordTimeTolerance`.

service.set_ledger_time_record_time_tolerance

Summary: Update the *ledgerTimeRecordTimeTolerance* in the dynamic domain parameters.

Arguments:

- `newLedgerTimeRecordTimeTolerance`: com.digitalasset.canton.config.NonNegativeFiniteDuration
- `force`: Boolean

Description: If it would be insecure to perform the change immediately, the command will block and wait until it is secure to perform the change. The command will block for at most twice of `newLedgerTimeRecordTimeTolerance`. If the domain does not support `mediatorDeduplicationTimeout`, the method will update `ledgerTimeRecordTimeTolerance` immediately without blocking. The method will fail if `mediatorDeduplicationTimeout` is less than twice of `newLedgerTimeRecordTimeTolerance`. Do not modify domain parameters concurrently while running this command, because the command may override concurrent changes. force: update `ledgerTimeRecordTimeTolerance` immediately without blocking. This is safe to do during domain bootstrapping and in test environments, but should not be done in operational production systems..

service.set_max_inbound_message_size

Summary: Try to update the max rate per participant for the domain

Arguments:

- `maxRequestSize`: com.digitalasset.canton.config.RequireTypes.NonNegativeInt
- `force`: Boolean

Description: If the max request size is dynamic, update the value. The update won't have any effect unless the sequencer server is restarted. If the max request size is not dynamic (i.e., if the domain is running on protocol version lower than 4), then it will throw an error.

service.set_max_rate_per_participant

Summary: Try to update the max rate per participant for the domain

Arguments:

- `maxRatePerParticipant`: com.digitalasset.canton.config.RequireTypes.NonNegativeInt

Description: If the max rate per participant is dynamic, update the value. If the max rate per participant is not dynamic (i.e., if the domain is running on protocol version lower than 4), then it will throw an error.

service.set_max_request_size

Summary: Try to update the max rate per participant for the domain

Arguments:

- `maxRequestSize`: com.digitalasset.canton.config.RequireTypes.NonNegativeInt
- `force`: Boolean

Description: If the max request size is dynamic, update the value. The update won't have any effect unless the sequencer server is restarted. If the max request size is not dynamic (i.e., if the domain is running on protocol version lower than 4), then it will throw an error.

service.set_mediator_deduplication_timeout

Summary: Update the mediator deduplication timeout

Arguments:

- `newMediatorDeduplicationTimeout`: com.digitalasset.canton.config.NonNegativeFiniteDuration

Description: The method will fail: - if the domain does not support the `mediatorDeduplicationTimeout` parameter, - if the new value of `mediatorDeduplicationTimeout` is less than twice the value of `ledgerTimeRecordTimeTolerance`.

service.set_reconciliation_interval

Summary: Try to update the reconciliation interval for the domain

Arguments:

- `newReconciliationInterval`: com.digitalasset.canton.config.PositiveDurationSeconds

Description: If the reconciliation interval is dynamic, update the value. If the reconciliation interval is not dynamic (i.e., if the domain is running on protocol version lower than 4), then it will throw an error.

service.update_dynamic_domain_parameters

Summary: Update the Dynamic Domain Parameters for the domain

Arguments:

- `modifier`: com.digitalasset.canton.admin.api.client.data.DynamicDomainParameters => com.digitalasset.canton.admin.api.client.data.DynamicDomainParameters
- `force`: Boolean

Description: force: Enable potentially dangerous changes. Required to increase `ledgerTimeRecordTimeTolerance`. Use `set_ledger_time_record_time_tolerance_securely` to securely increase `ledgerTimeRecordTimeTolerance`.

service.update_dynamic_parameters

Summary: Update the Dynamic Domain Parameters for the domain

Arguments:

- `modifier`: com.digitalasset.canton.admin.api.client.data.DynamicDomainParameters => com.digitalasset.canton.admin.api.client.data.DynamicDomainParameters
- `force`: Boolean

Description: force: Enable potentially dangerous changes. Required to increase `ledgerTimeRecordTimeTolerance`. Use `set_ledger_time_record_time_tolerance_securely` to securely increase `ledgerTimeRecordTimeTolerance`.

Topology Administration

Topology commands run on the domain topology manager immediately affect the topology state of the domain, which means that all changes are immediately pushed to the connected participants.

topology.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

topology.init_id

Summary: Initialize the node with a unique identifier

Arguments:

- `identifier`: com.digitalasset.canton.topology.Identifier

- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint

Return type:

- com.digitalasset.canton.topology.UniqueIdentifier

Description: Every node in Canton is identified using a unique identifier, which is composed of a user-chosen string and the fingerprint of a signing key. The signing key is the root key defining a so-called namespace, where the signing key has the ultimate control over issuing new identifiers. During initialisation, we have to pick such a unique identifier. By default, initialisation happens automatically, but it can be turned off by setting the auto-init option to false. Automatic node initialisation is usually turned off to preserve the identity of a participant or domain node (during major version upgrades) or if the topology transactions are managed through a different topology manager than the one integrated into this node.

topology.load_transaction

Summary: Upload signed topology transaction

Arguments:

- `bytes`: com.google.protobuf.ByteString

Description: Topology transactions can be issued with any topology manager. In some cases, such transactions need to be copied manually between nodes. This function allows for uploading previously exported topology transaction into the authorized store (which is the name of the topology managers transaction store.

topology.stores.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

topology.stores.list

Summary: List available topology stores

Return type:

- Seq[String]

Description: Topology transactions are stored in these stores. There are the following stores: Authorized - The authorized store is the store of a topology manager. Updates to the topology state are made by adding new transactions to the Authorized store. Both the participant and the domain nodes topology manager have such a store. A participant node will distribute all the content in the Authorized store to the domains it is connected to. The domain node will distribute the content of the Authorized store through the sequencer to the domain members in order to create the authoritative topology state on a domain (which is stored in the store named using the domain-id), such that every domain member will have the same view on the topology state on a particular domain. <domain-id> - The domain store is the authorized topology state on a domain. A participant has one store for each domain it is connected to. The domain has exactly one store with its domain-id. Requested - A domain can be configured such that when participant tries to register a topology transaction with the domain, the transaction is placed into the Requested store such that it can be analysed and processed with user defined process.

topology.namespace_delegations.authorize

Summary: Change namespace delegation

Arguments:

- `ops`: com.digitalasset.canton.topology.transaction.TopologyChangeOp
- `namespace`: com.digitalasset.canton.crypto.Fingerprint
- `authorizedKey`: com.digitalasset.canton.crypto.Fingerprint

- `isRootDelegation`: Boolean
- `signedBy`: Option[com.digitalasset.canton.crypto.Fingerprint]
- `synchronize`: Option[com.digitalasset.canton.config.NonNegativeDuration]
- `force`: Boolean

Return type:

- com.google.protobuf.ByteString

Description: Delegates the authority to authorize topology transactions in a certain namespace to a certain key. The keys are referred to using their fingerprints. They need to be either locally generated or have been previously imported. ops: Either Add or Remove the delegation. namespace: The namespace whose authorization authority is delegated. signedBy: Optional fingerprint of the authorizing key. The authorizing key needs to be either the authorizedKey for root certificates. Otherwise, the signedBy key needs to refer to a previously authorized key, which means that we use the signedBy key to refer to a locally available CA. authorizedKey: Fingerprint of the key to be authorized. If signedBy equals authorizedKey, then this transaction corresponds to a self-signed root certificate. If the keys differ, then we get an intermediate CA. isRootDelegation: If set to true (default = false), the authorized key will be allowed to issue NamespaceDelegations. synchronize: Synchronize timeout can be used to ensure that the state has been propagated into the node

topology.namespace_delegations.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

topology.namespace_delegations.list

Summary: List namespace delegation transactions

Arguments:

- `filterStore`: String
- `useStateStore`: Boolean
- `timeQuery`: com.digitalasset.canton.topology.store.TimeQuery
- `operation`: Option[com.digitalasset.canton.topology.transaction.TopologyChangeOp]
- `filterNamespace`: String
- `filterSigningKey`: String
- `filterTargetKey`: Option[com.digitalasset.canton.crypto.Fingerprint]
- `protocolVersion`: Option[String]

Return type:

- Seq[com.digitalasset.canton.admin.api.client.data.ListNamespaceDelegationResult]

Description: List the namespace delegation transaction present in the stores. Namespace delegations are topology transactions that permission a key to issue topology transactions within a certain namespace. filterStore: Filter for topology stores starting with the given filter string (Authorized, <domain-id>, Requested) useStateStore: If true (default), only properly authorized transactions that are part of the state will be selected. timeQuery: The time query allows to customize the query by time. The following options are supported: TimeQuery.HeadState (default): The most recent known state. TimeQuery.Snapshot(ts): The state at a certain point in time. TimeQuery.Range(fromO, toO): Time-range of when the transaction was added to the store operation: Optionally, what type of operation the transaction should have. State store only has Add . filterSigningKey: Filter for transactions that are authorized with a key that starts with the given filter string. filterNamespace: Filter for namespaces starting with the given filter string. filterTargetKey: Filter for

namespaces delegations for the given target key. protocolVersion: Export the topology transactions in the optional protocol version.

topology.identifier_delegations.authorize

Summary: Change identifier delegation

Arguments:

- `ops`: com.digitalasset.canton.topology.transaction.TopologyChangeOp
- `identifier`: com.digitalasset.canton.topology.UniqueIdentifier
- `authorizedKey`: com.digitalasset.canton.crypto.Fingerprint
- `signedBy`: Option[com.digitalasset.canton.crypto.Fingerprint]
- `synchronize`: Option[com.digitalasset.canton.config.NonNegativeDuration]

Return type:

- com.google.protobuf.ByteString

Description: Delegates the authority of a certain identifier to a certain key. This corresponds to a normal certificate which binds identifier to a key. The keys are referred to using their fingerprints. They need to be either locally generated or have been previously imported. ops: Either Add or Remove the delegation. signedBy: Refers to the optional fingerprint of the authorizing key which in turn refers to a specific, locally existing certificate. authorizedKey: Fingerprint of the key to be authorized. synchronize: Synchronize timeout can be used to ensure that the state has been propagated into the node

topology.identifier_delegations.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

topology.identifier_delegations.list

Summary: List identifier delegation transactions

Arguments:

- `filterStore`: String
- `useStateStore`: Boolean
- `timeQuery`: com.digitalasset.canton.topology.store.TimeQuery
- `operation`: Option[com.digitalasset.canton.topology.transaction.TopologyChangeOp]
- `filterUid`: String
- `filterSigningKey`: String
- `filterTargetKey`: Option[com.digitalasset.canton.crypto.Fingerprint]
- `protocolVersion`: Option[String]

Return type:

- Seq[com.digitalasset.canton.admin.api.client.data.ListIdentifierDelegationResult]

Description: List the identifier delegation transaction present in the stores. Identifier delegations are topology transactions that permission a key to issue topology transactions for a certain unique identifier. filterStore: Filter for topology stores starting with the given filter string (Authorized, <domain-id>, Requested) useStateStore: If true (default), only properly authorized transactions that are part of the state will be selected. timeQuery: The time query allows to customize the query by time. The following options are supported: TimeQuery.HeadState (default): The most recent known state. TimeQuery.Snapshot(ts): The state at a certain point in time. TimeQuery.Range(fromO, toO): Time-range of when the transaction was added to the store operation: Optionally, what type of operation the transaction should have. State store only has Add . filterSigningKey: Filter for transactions that are authorized with a key that starts with the given filter string. filterUid: Filter for

unique identifiers starting with the given filter string. protocolVersion: Export the topology transactions in the optional protocol version.

topology.owner_to_key_mappings.authorize

Summary: Change an owner to key mapping

Arguments:

- `ops`: com.digitalasset.canton.topology.transaction.TopologyChangeOp
- `keyOwner`: com.digitalasset.canton.topology.KeyOwner
- `key`: com.digitalasset.canton.crypto.Fingerprint
- `purpose`: com.digitalasset.canton.crypto.KeyPurpose
- `signedBy`: Option[com.digitalasset.canton.crypto.Fingerprint]
- `synchronize`: Option[com.digitalasset.canton.config.NonNegativeDuration]
- `force`: Boolean

Return type:

- com.google.protobuf.ByteString

Description: Change a owner to key mapping. A key owner is anyone in the system that needs a key-pair known to all members (participants, mediator, sequencer, topology manager) of a domain. ops: Either Add or Remove the key mapping update. signedBy: Optional fingerprint of the authorizing key which in turn refers to a specific, locally existing certificate. ownerType: Role of the following owner (Participant, Sequencer, Mediator, DomainTopologyManager) owner: Unique identifier of the owner. key: Fingerprint of key purposes: The purposes of the owner to key mapping. force: removing the last key is dangerous and must therefore be manually forced synchronize: Synchronize timeout can be used to ensure that the state has been propagated into the node

topology.owner_to_key_mappings.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

topology.owner_to_key_mappings.list

Summary: List owner to key mapping transactions

Arguments:

- `filterStore`: String
- `useStateStore`: Boolean
- `timeQuery`: com.digitalasset.canton.topology.store.TimeQuery
- `operation`: Option[com.digitalasset.canton.topology.transaction.TopologyChangeOp]
- `filterKeyOwnerType`: Option[com.digitalasset.canton.topology.KeyOwnerCode]
- `filterKeyOwnerUid`: String
- `filterKeyPurpose`: Option[com.digitalasset.canton.crypto.KeyPurpose]
- `filterSigningKey`: String
- `protocolVersion`: Option[String]

Return type:

- Seq[com.digitalasset.canton.admin.api.client.data.ListOwnerToKeyMappingResult]

Description: List the owner to key mapping transactions present in the stores. Owner to key mappings are topology transactions defining that a certain key is used by a certain key owner. Key owners are participants, sequencers, mediators and domains. filterStore: Filter for topology stores starting with the given filter string (Authorized, <domain-id>, Requested) useStateStore: If true (default), only properly authorized transactions that are

part of the state will be selected. timeQuery: The time query allows to customize the query by time. The following options are supported: TimeQuery.HeadState (default): The most recent known state. TimeQuery.Snapshot(ts): The state at a certain point in time. TimeQuery.Range(fromO, toO): Time-range of when the transaction was added to the store operation: Optionally, what type of operation the transaction should have. State store only has Add . filterSigningKey: Filter for transactions that are authorized with a key that starts with the given filter string. filterKeyOwnerType: Filter for a particular type of key owner (KeyOwnerCode). filterKeyOwnerUid: Filter for key owners unique identifier starting with the given filter string. filterKeyPurpose: Filter for keys with a particular purpose (Encryption or Signing) protocolVersion: Export the topology transactions in the optional protocol version.

topology.owner_to_key_mappings.rotate_key

Summary: Rotate the key for an owner to key mapping

Arguments:

- `nodeInstance`: com.digitalasset.canton.console.InstanceReferenceCommon
- `owner`: com.digitalasset.canton.topology.KeyOwner
- `currentKey`: com.digitalasset.canton.crypto.PublicKey
- `newKey`: com.digitalasset.canton.crypto.PublicKey

Description: Rotates the key for an existing owner to key mapping by issuing a new owner to key mapping with the new key and removing the previous owner to key mapping with the previous key. nodeInstance: The node instance that is used to verify that both current and new key pertain to this node. This avoids conflicts when there are different nodes with the same uuid (i.e., multiple sequencers). owner: The owner of the owner to key mapping currentKey: The current public key that will be rotated newKey: The new public key that has been generated

topology.party_to_participant_mappings.authorize (Preview)

Summary: Change party to participant mapping

Arguments:

- `ops`: com.digitalasset.canton.topology.transaction.TopologyChangeOp
- `party`: com.digitalasset.canton.topology.PartyId
- `participant`: com.digitalasset.canton.topology.ParticipantId
- `side`: com.digitalasset.canton.topology.transaction.RequestSide
- `permission`: com.digitalasset.canton.topology.transaction.ParticipantPermission
- `signedBy`: Option[com.digitalasset.canton.crypto.Fingerprint]
- `synchronize`: Option[com.digitalasset.canton.config.NonNegativeDuration]
- `replaceExisting`: Boolean
- `force`: Boolean

Return type:

- com.google.protobuf.ByteString

Description: Change the association of a party to a participant. If both identifiers are in the same namespace, then the request-side is Both. If they differ, then we need to say whether the request comes from the party (RequestSide.From) or from the participant (RequestSide.To). And, we need the matching request of the other side. Please note that this is a preview feature due to the fact that inhomogeneous topologies can not yet be properly represented on the Ledger API. ops: Either Add or Remove the mapping signedBy: Refers to the optional fingerprint of the authorizing key which in turn refers to a specific, locally existing certificate. party: The unique identifier of the party we want to map to a participant. participant: The unique identifier of the participant to which the party is supposed to be mapped. side: The request side (RequestSide.From if we the transaction is from the

perspective of the party, RequestSide.To from the participant.) privilege: The privilege of the given participant which allows us to restrict an association (e.g. Confirmation or Observation). replaceExisting: If true (default), replace any existing mapping with the new setting synchronize: Synchronize timeout can be used to ensure that the state has been propagated into the node

topology.party_to_participant_mappings.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

topology.party_to_participant_mappings.list

Summary: List party to participant mapping transactions

Arguments:

- `filterStore`: String
- `useStateStore`: Boolean
- `timeQuery`: com.digitalasset.canton.topology.store.TimeQuery
- `operation`: Option[com.digitalasset.canton.topology.transaction.TopologyChangeOp]
- `filterParty`: String
- `filterParticipant`: String
- `filterRequestSide`: Option[com.digitalasset.canton.topology.transaction.RequestSide]
- `filterPermission`: Option[com.digitalasset.canton.topology.transaction.ParticipantPermission]
- `filterSigningKey`: String
- `protocolVersion`: Option[String]

Return type:

- Seq[com.digitalasset.canton.admin.api.client.data.ListPartyToParticipantResult]

Description: List the party to participant mapping transactions present in the stores. Party to participant mappings are topology transactions used to allocate a party to a certain participant. The same party can be allocated on several participants with different privileges. A party to participant mapping has a request-side that identifies whether the mapping is authorized by the party, by the participant or by both. In order to have a party be allocated to a given participant, we therefore need either two transactions (one with RequestSide.From, one with RequestSide.To) or one with RequestSide.Both. filterStore: Filter for topology stores starting with the given filter string (Authorized, <domain-id>, Requested) useStateStore: If true (default), only properly authorized transactions that are part of the state will be selected. timeQuery: The time query allows to customize the query by time. The following options are supported: TimeQuery.HeadState (default): The most recent known state. TimeQuery.Snapshot(ts): The state at a certain point in time. TimeQuery.Range(fromO, toO): Time-range of when the transaction was added to the store operation: Optionally, what type of operation the transaction should have. State store only has Add . filterSigningKey: Filter for transactions that are authorized with a key that starts with the given filter string. filterParty: Filter for parties starting with the given filter string. filterParticipant: Filter for participants starting with the given filter string. filterRequestSide: Optional filter for a particular request side (Both, From, To). protocolVersion: Export the topology transactions in the optional protocol version.

topology.participant_domain_states.active

Summary: Returns true if the given participant is currently active on the given domain

Arguments:

- domainId: com.digitalasset.canton.topology.DomainId
- participantId: com.digitalasset.canton.topology.ParticipantId

Return type:

- Boolean

Description: Active means that the participant has been granted at least observation rights on the domain and that the participant has registered a domain trust certificate

topology.participant_domain_states.authorize

Summary: Change participant domain states

Arguments:

- ops: com.digitalasset.canton.topology.transaction.TopologyChangeOp
- domain: com.digitalasset.canton.topology.DomainId
- participant: com.digitalasset.canton.topology.ParticipantId
- side: com.digitalasset.canton.topology.transaction.RequestSide
- permission: com.digitalasset.canton.topology.transaction.ParticipantPermission
- trustLevel: com.digitalasset.canton.topology.transaction.TrustLevel
- signedBy: Option[com.digitalasset.canton.crypto.Fingerprint]
- synchronize: Option[com.digitalasset.canton.config.NonNegativeDuration]
- replaceExisting: Boolean

Return type:

- com.google.protobuf.ByteString

Description: Change the association of a participant to a domain. In order to activate a participant on a domain, we need both authorisation: the participant authorising its uid to be present on a particular domain and the domain to authorise the presence of a participant on said domain. If both identifiers are in the same namespace, then the request-side can be Both. If they differ, then we need to say whether the request comes from the domain (RequestSide.From) or from the participant (RequestSide.To). And, we need the matching request of the other side. ops: Either Add or Remove the mapping signedBy: Refers to the optional fingerprint of the authorizing key which in turn refers to a specific, locally existing certificate. domain: The unique identifier of the domain we want the participant to join. participant: The unique identifier of the participant. side: The request side (RequestSide.From if we the transaction is from the perspective of the domain, RequestSide.To from the participant.) permission: The privilege of the given participant which allows us to restrict an association (e.g. Confirmation or Observation). Will use the lower of if different between To/From. trustLevel: The trust level of the participant on the given domain. Will use the lower of if different between To/From. replaceExisting: If true (default), replace any existing mapping with the new setting synchronize: Synchronize timeout can be used to ensure that the state has been propagated into the node

topology.participant_domain_states.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- methodName: String

topology.participant_domain_states.list

Summary: List participant domain states

Arguments:

- filterStore: String
- useStateStore: Boolean
- timeQuery: com.digitalasset.canton.topology.store.TimeQuery
- operation: Option[com.digitalasset.canton.topology.transaction.Topology-

ChangeOp]
- `filterDomain`: String
- `filterParticipant`: String
- `filterSigningKey`: String
- `protocolVersion`: Option[String]

Return type:
- Seq[com.digitalasset.canton.admin.api.client.data.ListParticipantDomainStateResult]

Description: List the participant domain transactions present in the stores. Participant domain states are topology transactions used to permission a participant on a given domain. A participant domain state has a request-side that identifies whether the mapping is authorized by the participant (From), by the domain (To) or by both (Both). In order to use a participant on a domain, both have to authorize such a mapping. This means that by authorizing such a topology transaction, a participant acknowledges its presence on a domain, whereas a domain permissions the participant on that domain. filterStore: Filter for topology stores starting with the given filter string (Authorized, <domain-id>, Requested) useStateStore: If true (default), only properly authorized transactions that are part of the state will be selected. timeQuery: The time query allows to customize the query by time. The following options are supported: TimeQuery.HeadState (default): The most recent known state. TimeQuery.Snapshot(ts): The state at a certain point in time. TimeQuery.Range(fromO, toO): Time-range of when the transaction was added to the store operation: Optionally, what type of operation the transaction should have. State store only has Add . filterSigningKey: Filter for transactions that are authorized with a key that starts with the given filter string. filterDomain: Filter for domains starting with the given filter string. filterParticipant: Filter for participants starting with the given filter string. protocolVersion: Export the topology transactions in the optional protocol version.

topology.vetted_packages.authorize

Summary: Change package vettings

Arguments:
- `ops`: com.digitalasset.canton.topology.transaction.TopologyChangeOp
- `participant`: com.digitalasset.canton.topology.ParticipantId
- `packageIds`: Seq[com.daml.lf.data.Ref.PackageId]
- `signedBy`: Option[com.digitalasset.canton.crypto.Fingerprint]
- `synchronize`: Option[com.digitalasset.canton.config.NonNegativeDuration]
- `force`: Boolean

Return type:
- com.google.protobuf.ByteString

Description: A participant will only process transactions that reference packages that all involved participants have vetted previously. Vetting is done by registering a respective topology transaction with the domain, which can then be used by other participants to verify that a transaction is only using vetted packages. Note that all referenced and dependent packages must exist in the package store. By default, only vetting transactions adding new packages can be issued. Removing package vettings and issuing package vettings for other participants (if their identity is controlled through this participants topology manager) or for packages that do not exist locally can only be run using the force = true flag. However, these operations are dangerous and can lead to the situation of a participant being unable to process transactions. ops: Either Add or Remove the vetting. participant: The unique identifier of the participant that is vetting the package. packageIds: The lf-package ids to be vetted. signedBy: Refers to the fingerprint of the authorizing key which in turn must be authorized by a valid, locally existing certificate. If none is given, a key is automatically determined. synchronize: Synchronize timeout can be used to en-

sure that the state has been propagated into the node force: Flag to enable dangerous operations (default false). Great power requires great care.

topology.vetted_packages.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

topology.vetted_packages.list

Summary: List package vetting transactions

Arguments:

- `filterStore`: String
- `useStateStore`: Boolean
- `timeQuery`: com.digitalasset.canton.topology.store.TimeQuery
- `operation`: Option[com.digitalasset.canton.topology.transaction.TopologyChangeOp]
- `filterParticipant`: String
- `filterSigningKey`: String
- `protocolVersion`: Option[String]

Return type:

- Seq[com.digitalasset.canton.admin.api.client.data.ListVettedPackagesResult]

Description: List the package vetting transactions present in the stores. Participants must vet Daml packages and submitters must ensure that the receiving participants have vetted the package prior to submitting a transaction (done automatically during submission and validation). Vetting is done by authorizing such topology transactions and registering with a domain. filterStore: Filter for topology stores starting with the given filter string (Authorized, <domain-id>, Requested) useStateStore: If true (default), only properly authorized transactions that are part of the state will be selected. timeQuery: The time query allows to customize the query by time. The following options are supported: TimeQuery.HeadState (default): The most recent known state. TimeQuery.Snapshot(ts): The state at a certain point in time. TimeQuery.Range(fromO, toO): Time-range of when the transaction was added to the store operation: Optionally, what type of operation the transaction should have. State store only has Add . filterSigningKey: Filter for transactions that are authorized with a key that starts with the given filter string. filterParticipant: Filter for participants starting with the given filter string. protocolVersion: Export the topology transactions in the optional protocol version.

topology.all.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

topology.all.list

Summary: List all transaction

Arguments:

- `filterStore`: String
- `useStateStore`: Boolean
- `timeQuery`: com.digitalasset.canton.topology.store.TimeQuery
- `operation`: Option[com.digitalasset.canton.topology.transaction.TopologyChangeOp]
- `filterAuthorizedKey`: Option[com.digitalasset.canton.crypto.Fingerprint]

- `protocolVersion`: Option[String]

Return type:

- com.digitalasset.canton.topology.store.StoredTopologyTransactions[com.digitalasset.canton.topology.transaction.TopologyChangeOp]

Description: List all topology transactions in a store, independent of the particular type. This method is useful for exporting entire states. filterStore: Filter for topology stores starting with the given filter string (Authorized, <domain-id>, Requested) useStateStore: If true (default), only properly authorized transactions that are part of the state will be selected. timeQuery: The time query allows to customize the query by time. The following options are supported: TimeQuery.HeadState (default): The most recent known state. TimeQuery.Snapshot(ts): The state at a certain point in time. TimeQuery.Range(fromO, toO): Time-range of when the transaction was added to the store operation: Optionally, what type of operation the transaction should have. State store only has Add . filterAuthorizedKey: Filter the topology transactions by the key that has authorized the transactions. protocolVersion: Export the topology transactions in the optional protocol version.

topology.all.renew

Summary: Renew all topology transactions that have been authorized with a previous key using a new key

Arguments:

- `filterAuthorizedKey`: com.digitalasset.canton.crypto.Fingerprint
- `authorizeWith`: com.digitalasset.canton.crypto.Fingerprint

Description: Finds all topology transactions that have been authorized by *filterAuthorizedKey* and renews those topology transactions by authorizing them with the new key *authorizeWith*. filterAuthorizedKey: Filter the topology transactions by the key that has authorized the transactions. authorizeWith: The key to authorize the renewed topology transactions.

1.28.3.5 Domain Manager Administration Commands

clear_cache (Testing)

Summary: Clear locally cached variables

Description: Some commands cache values on the client side. Use this command to explicitly clear the caches of these values.

config

Summary: Returns the domain configuration

Return type:

- com.digitalasset.canton.domain.config.DomainManagerConfig

help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

id

Summary: Yields the globally unique id of this domain. Throws an exception, if the id has not yet been allocated (e.g., the domain has not yet been started).

Return type:

- com.digitalasset.canton.topology.DomainId

is_initialized

Summary: Check if the local instance is running and is fully initialized
Return type:
- Boolean

is_running
Summary: Check if the local instance is running
Return type:
- Boolean

start
Summary: Start the instance

stop
Summary: Stop the instance

Setup

setup.authorize_mediator
Summary: Authorize external Mediator node.
Arguments:
- `mediatorId`: com.digitalasset.canton.topology.MediatorId

Description: Use this command to reinstigate an external mediator node that has been offboarded via offboard_mediator.

setup.bootstrap_domain
Summary: Bootstrap domain
Arguments:
- `sequencers`: Seq[com.digitalasset.canton.console.SequencerNodeReference]
- `mediators`: Seq[com.digitalasset.canton.console.MediatorReference]

Description: Use this command to bootstrap the domain with an initial set of external sequencer(s) and external mediator(s). Note that you only need to call this once, however it is safe to call it again if necessary in case something went wrong and this needs to be retried.

setup.help
Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

setup.init
Summary: Initialize domain
Arguments:
- `sequencerConnection`: com.digitalasset.canton.sequencing.SequencerConnection

Description: This command triggers domain initialization and should be called once the initial topology data has been authorized and sequenced. This is called as part of the setup.bootstrap command, so you are unlikely to need to call this directly.

setup.init
Summary: Initialize domain
Arguments:
- `sequencerConnections`: com.digitalasset.canton.sequencing.SequencerConnections

Description: This command triggers domain initialization and should be called once the initial topology data has been authorized and sequenced. This is called as part of the setup.bootstrap command, so you are unlikely to need to call this directly.

setup.offboard_mediator

Summary: Offboard external Mediator node.

Arguments:

- `mediatorId`: com.digitalasset.canton.topology.MediatorId
- `force`: Boolean

Description: Use this command to offboard an onboarded external mediator node. It removes the topology transaction that authorizes the given mediator ID to act as a mediator on the domain. If you afterwards want to authorize an offboarded mediator again, use authorize_mediator. You must apply force to offboard the last mediator of a domain.

setup.onboard_mediator

Summary: Onboard external Mediator node.

Arguments:

- `mediator`: com.digitalasset.canton.console.MediatorReference
- `sequencerConnections`: Seq[com.digitalasset.canton.console.InstanceReferenceWithSequencerConnection]

Description: Use this command to onboard an external mediator node. If you're bootstrapping a domain with external sequencer(s) and this is the initial mediator, then use setup.bootstrap_domain instead. For adding additional external mediators or onboard an external mediator with a domain that runs a single embedded sequencer, use this command.Note that you only need to call this once.

setup.onboard_new_sequencer

Summary: Dynamically onboard new Sequencer node.

Arguments:

- `initialSequencer`: com.digitalasset.canton.console.SequencerNodeReference
- `newSequencer`: com.digitalasset.canton.console.SequencerNodeReference

Return type:

- com.digitalasset.canton.crypto.PublicKey

Description: Use this command to dynamically onboard a new sequencer node that's not part of the initial set of sequencer nodes. Do not use this for database sequencers.

Health

health.active

Summary: Check if the node is running and is the active instance (mediator, participant)

Return type:

- Boolean

health.dump

Summary: Creates a zip file containing diagnostic information about the canton process running this node

Arguments:

- `outputFile`: better.files.File
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration
- `chunkSize`: Option[Int]

Return type:

- String

health.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

health.initialized

Summary: Returns true if node has been initialized.

Return type:

- Boolean

health.running

Summary: Check if the node is running

Return type:

- Boolean

health.status

Summary: Get human (and machine) readable status info

Return type:

- com.digitalasset.canton.health.admin.data.NodeStatus[S]

health.wait_for_identity

Summary: Wait for the node to have an identity

Description: This is specifically useful for the Domain Manager which needs its identity to be ready for bootstrapping, but for which we can't rely on wait_for_initialized() because it will be initialized only after being bootstrapped.

health.wait_for_initialized

Summary: Wait for the node to be initialized

health.wait_for_running

Summary: Wait for the node to be running

Database

db.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

db.migrate

Summary: Migrates the instance's database if using a database storage

db.repair_migration

Summary: Only use when advised - repairs the database migration of the instance's database

Arguments:

- `force`: Boolean

Description: In some rare cases, we change already applied database migration files in a new release and the repair command resets the checksums we use to ensure that in general already applied migration files have not been changed. You should only use *db.repair_migration* when advised and otherwise use it at your own risk - in the worst case running it may lead to data corruption when an incompatible database migration (one that

should be rejected because the already applied database migration files have changed) is subsequently falsely applied.

Sequencer Connection

sequencer_connection.get

Summary: Get Sequencer Connection
Return type:
- Option[com.digitalasset.canton.sequencing.SequencerConnections]

Description: Use this command to get the currently configured sequencer connection details for this sequencer client. If this node has not yet been initialized, this will return None.

sequencer_connection.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

sequencer_connection.modify

Summary: Modify Default Sequencer Connection
Arguments:
- `modifier`: com.digitalasset.canton.sequencing.SequencerConnection => com.digitalasset.canton.sequencing.SequencerConnection

Description: Modify sequencer connection details for this sequencer client node, by passing a modifier function that operates on the existing default connection.

sequencer_connection.modify_connections

Summary: Modify Sequencer Connections
Arguments:
- `modifier`: com.digitalasset.canton.sequencing.SequencerConnections => com.digitalasset.canton.sequencing.SequencerConnections

Description: Modify sequencer connection details for this sequencer client node, by passing a modifier function that operates on the existing connection configuration.

sequencer_connection.set

Summary: Set Sequencer Connection
Arguments:
- `connection`: com.digitalasset.canton.sequencing.SequencerConnection

Description: Set new sequencer connection details for this sequencer client node. This will replace any pre-configured connection details. This command will only work after the node has been initialized.

sequencer_connection.set

Summary: Set Sequencer Connection
Arguments:
- `connections`: com.digitalasset.canton.sequencing.SequencerConnections

Description: Set new sequencer connection details for this sequencer client node. This will replace any pre-configured connection details. This command will only work after the node has been initialized.

Key Administration

keys.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

keys.public.download

Summary: Download public key
Arguments:
- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint
- `protocolVersion`: com.digitalasset.canton.version.ProtocolVersion

Return type:
- com.google.protobuf.ByteString

keys.public.download_to

Summary: Download public key and save it to a file
Arguments:
- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint
- `outputFile`: String
- `protocolVersion`: com.digitalasset.canton.version.ProtocolVersion

keys.public.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

keys.public.list

Summary: List public keys in registry
Arguments:
- `filterFingerprint`: String
- `filterContext`: String

Return type:
- Seq[com.digitalasset.canton.crypto.PublicKeyWithName]

Description: Returns all public keys that have been added to the key registry. Optional arguments can be used for filtering.

keys.public.list_by_owner

Summary: List keys for given keyOwner.
Arguments:
- `keyOwner`: com.digitalasset.canton.topology.KeyOwner
- `filterDomain`: String
- `asOf`: Option[java.time.Instant]
- `limit`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:
- Seq[com.digitalasset.canton.admin.api.client.data.ListKeyOwnersResult]

Description: This command is a convenience wrapper for *list_key_owners*, taking an explicit keyOwner as search argument. The response includes the public keys.

keys.public.list_owners

Summary: List active owners with keys for given search arguments.
Arguments:

- `filterKeyOwnerUid`: String
- `filterKeyOwnerType`: Option[com.digitalasset.canton.topology.KeyOwnerCode]
- `filterDomain`: String
- `asOf`: Option[java.time.Instant]
- `limit`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:

- Seq[com.digitalasset.canton.admin.api.client.data.ListKeyOwnersResult]

Description: This command allows deep inspection of the topology state. The response includes the public keys. Optional filterKeyOwnerType type can be 'ParticipantId.Code' , 'MediatorId.Code','SequencerId.Code', 'DomainTopologyManagerId.Code'.

keys.public.upload

Summary: Upload public key

Arguments:

- `filename`: String
- `name`: Option[String]

Return type:

- com.digitalasset.canton.crypto.Fingerprint

keys.public.upload

Summary: Upload public key

Arguments:

- `keyBytes`: com.google.protobuf.ByteString
- `name`: Option[String]

Return type:

- com.digitalasset.canton.crypto.Fingerprint

Description: Import a public key and store it together with a name used to provide some context to that key.

keys.secret.delete

Summary: Delete private key

Arguments:

- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint
- `force`: Boolean

keys.secret.download

Summary: Download key pair

Arguments:

- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint
- `protocolVersion`: com.digitalasset.canton.version.ProtocolVersion

Return type:

- com.google.protobuf.ByteString

keys.secret.download_to

Summary: Download key pair and save it to a file

Arguments:

- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint
- `outputFile`: String
- `protocolVersion`: com.digitalasset.canton.version.ProtocolVersion

keys.secret.generate_encryption_key

Summary: Generate new public/private key pair for encryption and store it in the vault

Arguments:

- `name`: String
- `scheme`: Option[com.digitalasset.canton.crypto.EncryptionKeyScheme]

Return type:

- com.digitalasset.canton.crypto.EncryptionPublicKey

Description: The optional name argument allows you to store an associated string for your convenience. The scheme can be used to select a key scheme and the default scheme is used if left unspecified.

keys.secret.generate_signing_key

Summary: Generate new public/private key pair for signing and store it in the vault

Arguments:

- `name`: String
- `scheme`: Option[com.digitalasset.canton.crypto.SigningKeyScheme]

Return type:

- com.digitalasset.canton.crypto.SigningPublicKey

Description: The optional name argument allows you to store an associated string for your convenience. The scheme can be used to select a key scheme and the default scheme is used if left unspecified.

keys.secret.get_wrapper_key_id

Summary: Get the wrapper key id that is used for the encrypted private keys store

Return type:

- String

keys.secret.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

keys.secret.list

Summary: List keys in private vault

Arguments:

- `filterFingerprint`: String
- `filterName`: String
- `purpose`: Set[com.digitalasset.canton.crypto.KeyPurpose]

Return type:

- Seq[com.digitalasset.canton.crypto.admin.grpc.PrivateKeyMetadata]

Description: Returns all public keys to the corresponding private keys in the key vault. Optional arguments can be used for filtering.

keys.secret.register_kms_encryption_key

Summary: Register the specified KMS encryption key in canton storing its public information in the vault

Arguments:

- `kmsKeyId`: String
- `name`: String

Return type:

- com.digitalasset.canton.crypto.EncryptionPublicKey

Description: The id for the KMS encryption key. The optional name argument allows you to store an associated string for your convenience.

keys.secret.register_kms_signing_key

Summary: Register the specified KMS signing key in canton storing its public information

in the vault
Arguments:
- `kmsKeyId`: String
- `name`: String

Return type:
- com.digitalasset.canton.crypto.SigningPublicKey

Description: The id for the KMS signing key. The optional name argument allows you to store an associated string for your convenience.

keys.secret.rotate_kms_node_key

Summary: Rotate a given node's keypair with a new pre-generated KMS keypair
Arguments:
- `fingerprint`: String
- `newKmsKeyId`: String

Return type:
- com.digitalasset.canton.crypto.PublicKey

Description: Rotates an existing encryption or signing key stored externally in a KMS with a pre-generated key. The fingerprint of the key we want to rotate. The id of the new KMS key (e.g. Resource Name).

keys.secret.rotate_node_key

Summary: Rotate a node's public/private key pair
Arguments:
- `fingerprint`: String
- `name`: Option[String]

Return type:
- com.digitalasset.canton.crypto.PublicKey

Description: Rotates an existing encryption or signing key. NOTE: A namespace root or intermediate signing key CANNOT be rotated by this command. The fingerprint of the key we want to rotate.

keys.secret.rotate_node_keys

Summary: Rotate the node's public/private key pairs
Description: For a participant node it rotates the signing and encryption key pair. For a domain or domain manager node it rotates the signing key pair as those nodes do not have an encryption key pair. For a sequencer or mediator node use *rotate_node_keys* with a domain manager reference as an argument. NOTE: Namespace root or intermediate signing keys are NOT rotated by this command.

keys.secret.rotate_wrapper_key

Summary: Change the wrapper key for encrypted private keys store
Arguments:
- `newWrapperKeyId`: String

Description: Change the wrapper key (e.g. AWS KMS key) being used to encrypt the private keys in the store. newWrapperKeyId: The optional new wrapper key id to be used. If the wrapper key id is empty Canton will generate a new key based on the current configuration.

keys.secret.upload

Summary: Upload a key pair
Arguments:
- `pairBytes`: com.google.protobuf.ByteString
- `name`: Option[String]

keys.secret.upload

Summary: Upload (load and import) a key pair from file
Arguments:
- `filename`: String
- `name`: Option[String]

Parties

parties.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

parties.list

Summary: List active parties, their active participants, and the participants' permissions on domains.
Arguments:
- `filterParty`: String
- `filterParticipant`: String
- `filterDomain`: String
- `asOf`: Option[java.time.Instant]
- `limit`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:
- Seq[com.digitalasset.canton.admin.api.client.data.ListPartiesResult]

Description: Inspect the parties known by this participant as used for synchronisation. The response is built from the timestamped topology transactions of each domain, excluding the authorized store of the given node. For each known party, the list of active participants and their permission on the domain for that party is given. filterParty: Filter by parties starting with the given string. filterParticipant: Filter for parties that are hosted by a participant with an id starting with the given string filterDomain: Filter by domains whose id starts with the given string. asOf: Optional timestamp to inspect the topology state at a given point in time. limit: Limit on the number of parties fetched (defaults to canton.parameters.console.default-limit). Example: participant1.parties.list(filterParty= alice)

Service

service.get_dynamic_domain_parameters

Summary: Get the Dynamic Domain Parameters configured for the domain
Return type:
- com.digitalasset.canton.admin.api.client.data.DynamicDomainParameters

service.get_max_rate_per_participant

Summary: Get the max rate per participant
Return type:
- com.digitalasset.canton.config.RequireTypes.NonNegativeInt

Description: Depending on the protocol version used on the domain, the value will be read either from the static domain parameters or the dynamic ones.

service.get_max_request_size

Summary: Get the max request size

Return type:
- com.digitalasset.canton.config.RequireTypes.NonNegativeInt

Description: Depending on the protocol version used on the domain, the value will be read either from the static domain parameters or the dynamic ones. This value is not necessarily the one used by the sequencer node because it requires a restart of the server to be taken into account.

service.get_mediator_deduplication_timeout

Summary: Get the mediator deduplication timeout

Return type:
- com.digitalasset.canton.config.NonNegativeFiniteDuration

Description: The method will fail, if the domain does not support the mediatorDeduplicationTimeout.

service.get_reconciliation_interval

Summary: Get the reconciliation interval configured for the domain

Return type:
- com.digitalasset.canton.config.PositiveDurationSeconds

Description: Depending on the protocol version used on the domain, the value will be read either from the static domain parameters or the dynamic ones.

service.get_static_domain_parameters

Summary: Get the Static Domain Parameters configured for the domain

Return type:
- com.digitalasset.canton.admin.api.client.data.StaticDomainParameters

service.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:
- `methodName`: String

service.list_accepted_agreements

Summary: List the accepted service agreements

Return type:
- Seq[com.digitalasset.canton.domain.service.ServiceAgreementAcceptance]

service.set_dynamic_domain_parameters

Summary: Set the Dynamic Domain Parameters configured for the domain

Arguments:
- `dynamicDomainParameters`: com.digitalasset.canton.admin.api.client.data.DynamicDomainParameters
- `force`: Boolean

Description: force: Enable potentially dangerous changes. Required to increase `ledgerTimeRecordTimeTolerance`. Use `set_ledger_time_record_time_tolerance` to securely increase `ledgerTimeRecordTimeTolerance`.

service.set_ledger_time_record_time_tolerance

Summary: Update the *ledgerTimeRecordTimeTolerance* in the dynamic domain parameters.

Arguments:
- `newLedgerTimeRecordTimeTolerance`: com.digitalasset.canton.config.NonNegativeFiniteDuration
- `force`: Boolean

Description: If it would be insecure to perform the change immediately, the command will block and wait until it is secure to perform the change. The command will block for at

most twice of `newLedgerTimeRecordTimeTolerance`. If the domain does not support `mediatorDeduplicationTimeout`, the method will update `ledgerTimeRecordTimeTolerance` immediately without blocking. The method will fail if `mediatorDeduplicationTimeout` is less than twice of `newLedgerTimeRecordTimeTolerance`. Do not modify domain parameters concurrently while running this command, because the command may override concurrent changes. force: update `ledgerTimeRecordTimeTolerance` immediately without blocking. This is safe to do during domain bootstrapping and in test environments, but should not be done in operational production systems..

service.set_max_inbound_message_size

Summary: Try to update the max rate per participant for the domain
Arguments:
- `maxRequestSize`: com.digitalasset.canton.config.RequireTypes.NonNegativeInt
- `force`: Boolean

Description: If the max request size is dynamic, update the value. The update won't have any effect unless the sequencer server is restarted. If the max request size is not dynamic (i.e., if the domain is running on protocol version lower than 4), then it will throw an error.

service.set_max_rate_per_participant

Summary: Try to update the max rate per participant for the domain
Arguments:
- `maxRatePerParticipant`: com.digitalasset.canton.config.RequireTypes.NonNegativeInt

Description: If the max rate per participant is dynamic, update the value. If the max rate per participant is not dynamic (i.e., if the domain is running on protocol version lower than 4), then it will throw an error.

service.set_max_request_size

Summary: Try to update the max rate per participant for the domain
Arguments:
- `maxRequestSize`: com.digitalasset.canton.config.RequireTypes.NonNegativeInt
- `force`: Boolean

Description: If the max request size is dynamic, update the value. The update won't have any effect unless the sequencer server is restarted. If the max request size is not dynamic (i.e., if the domain is running on protocol version lower than 4), then it will throw an error.

service.set_mediator_deduplication_timeout

Summary: Update the mediator deduplication timeout
Arguments:
- `newMediatorDeduplicationTimeout`: com.digitalasset.canton.config.NonNegativeFiniteDuration

Description: The method will fail: - if the domain does not support the `mediatorDeduplicationTimeout` parameter, - if the new value of `mediatorDeduplicationTimeout` is less than twice the value of `ledgerTimeRecordTimeTolerance`.

service.set_reconciliation_interval

Summary: Try to update the reconciliation interval for the domain
Arguments:
- `newReconciliationInterval`: com.digitalasset.canton.config.PositiveDurationSeconds

Description: If the reconciliation interval is dynamic, update the value. If the reconciliation interval is not dynamic (i.e., if the domain is running on protocol version lower than 4), then it will throw an error.

service.update_dynamic_domain_parameters

Summary: Update the Dynamic Domain Parameters for the domain
Arguments:
- `modifier`: com.digitalasset.canton.admin.api.client.data.DynamicDomainParameters => com.digitalasset.canton.admin.api.client.data.DynamicDomainParameters
- `force`: Boolean

Description: force: Enable potentially dangerous changes. Required to increase `ledgerTimeRecordTimeTolerance`. Use `set_ledger_time_record_time_tolerance_securely` to securely increase `ledgerTimeRecordTimeTolerance`.

service.update_dynamic_parameters

Summary: Update the Dynamic Domain Parameters for the domain
Arguments:
- `modifier`: com.digitalasset.canton.admin.api.client.data.DynamicDomainParameters => com.digitalasset.canton.admin.api.client.data.DynamicDomainParameters
- `force`: Boolean

Description: force: Enable potentially dangerous changes. Required to increase `ledgerTimeRecordTimeTolerance`. Use `set_ledger_time_record_time_tolerance_securely` to securely increase `ledgerTimeRecordTimeTolerance`.

Topology Administration

Same as *Domain Topology Administration*.

1.28.3.6 Sequencer Administration Commands

clear_cache (Testing)

Summary: Clear locally cached variables
Description: Some commands cache values on the client side. Use this command to explicitly clear the caches of these values.

config

Summary: Returns the sequencer configuration
Return type:
- com.digitalasset.canton.domain.sequencing.config.SequencerNodeConfig

help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

id

Summary: Yields the globally unique id of this sequencer. Throws an exception, if the id has not yet been allocated (e.g., the sequencer has not yet been started).
Return type:
- com.digitalasset.canton.topology.SequencerId

is_initialized

Summary: Check if the local instance is running and is fully initialized

Return type:
- Boolean

is_running

Summary: Check if the local instance is running
Return type:
- Boolean

start

Summary: Start the instance

stop

Summary: Stop the instance

Sequencer

sequencer.disable_member

Summary: Disable the provided member at the Sequencer that will allow any unread data for them to be removed
Arguments:
- `member`: com.digitalasset.canton.topology.Member

Description: This will prevent any client for the given member to reconnect the Sequencer and allow any unread/unacknowledged data they have to be removed. This should only be used if the domain operation is confident the member will never need to reconnect as there is no way to re-enable the member. To view members using the sequencer run *sequencer.status()*.

sequencer.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

sequencer.pruning.clear_schedule

Summary: Deactivate automatic pruning.

sequencer.pruning.force_prune

Summary: Force remove data from the Sequencer including data that may have not been read by offline clients
Arguments:
- `dryRun`: Boolean

Return type:
- String

Description: Will force pruning up until the default retention period by potentially disabling clients that have not yet read data we would like to remove. Disabling these clients will prevent them from ever reconnecting to the Domain so should only be used if the Domain operator is confident they can be permanently ignored. Run with *dryRun = true* to review a description of which clients will be disabled first. Run with *dryRun = false* to disable these clients and perform a forced pruning.

sequencer.pruning.force_prune_at

Summary: Force removing data from the Sequencer including data that may have not been read by offline clients up until the specified time
Arguments:

- `timestamp`: com.digitalasset.canton.data.CantonTimestamp
- `dryRun`: Boolean

Return type:

- String

Description: Similar to the above *force_prune* command but allows specifying the exact time at which to prune

sequencer.pruning.force_prune_with_retention_period

Summary: Force removing data from the Sequencer including data that may have not been read by offline clients up until a custom retention period

Arguments:

- `retentionPeriod`: scala.concurrent.duration.FiniteDuration
- `dryRun`: Boolean

Return type:

- String

Description: Similar to the above *force_prune* command but allows specifying a custom retention period

sequencer.pruning.get_schedule

Summary: Inspect the automatic pruning schedule.

Return type:

- Option[com.digitalasset.canton.admin.api.client.data.PruningSchedule]

Description: The schedule consists of a cron expression and max_duration and retention durations. The cron string indicates the points in time at which pruning should begin in the GMT time zone, and the maximum duration indicates how long from the start time pruning is allowed to run as long as pruning has not finished pruning up to the specified retention period. Returns *None* if no schedule has been configured via *set_schedule* or if *clear_schedule* has been invoked.

sequencer.pruning.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

sequencer.pruning.locate_pruning_timestamp

Summary: Obtain a timestamp at or near the beginning of sequencer state

Arguments:

- `index`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:

- Option[com.digitalasset.canton.data.CantonTimestamp]

Description: This command provides insight into the current state of sequencer pruning when called with the default value of *index* 1. When pruning the sequencer manually via *prune_at* and with the intent to prune in batches, specify a value such as 1000 to obtain a pruning timestamp that corresponds to the end of the batch.

sequencer.pruning.prune

Summary: Remove unnecessary data from the Sequencer up until the default retention point

Return type:

- String

Description: Removes unnecessary data from the Sequencer that is earlier than the default retention period. The default retention period is set in the configuration of the canton processing running this command under *parameters.retention-period-defaults.sequencer*.

This pruning command requires that data is read and acknowledged by clients before considering it safe to remove. If no data is being removed it could indicate that clients are not reading or acknowledging data in a timely fashion (typically due to nodes going offline for long periods). You have the option of disabling the members running on these nodes to allow removal of this data, however this will mean that they will be unable to reconnect to the domain in the future. To do this run *force_prune(dryRun = true)* to return a description of which members would be disabled in order to prune the Sequencer. If you are happy to disable the described clients then run *force_prune(dryRun = false)* to permanently remove their unread data. Once offline clients have been disabled you can continue to run *prune* normally.

sequencer.pruning.prune_at

Summary: Remove data that has been read up until the specified time

Arguments:

- `timestamp`: com.digitalasset.canton.data.CantonTimestamp

Return type:

- String

Description: Similar to the above *prune* command but allows specifying the exact time at which to prune. The command will fail if a client has not yet read and acknowledged some data up to the specified time.

sequencer.pruning.prune_with_retention_period

Summary: Remove data that has been read up until a custom retention period

Arguments:

- `retentionPeriod`: scala.concurrent.duration.FiniteDuration

Return type:

- String

Description: Similar to the above *prune* command but allows specifying a custom retention period

sequencer.pruning.set_cron

Summary: Modify the cron used by automatic pruning.

Arguments:

- `cron`: String

Description: The schedule is specified in cron format and refers to pruning start times in the GMT time zone. This call returns an error if no schedule has been configured via *set_schedule* or if automatic pruning has been disabled via *clear_schedule*. Additionally if at the time of this modification, pruning is actively running, a best effort is made to pause pruning and restart according to the new schedule. This allows for the case that the new schedule no longer allows pruning at the current time.

sequencer.pruning.set_max_duration

Summary: Modify the maximum duration used by automatic pruning.

Arguments:

- `maxDuration`: com.digitalasset.canton.config.PositiveDurationSeconds

Description: The *maxDuration* is specified as a positive duration and has at most per-second granularity. This call returns an error if no schedule has been configured via *set_schedule* or if automatic pruning has been disabled via *clear_schedule*. Additionally if at the time of this modification, pruning is actively running, a best effort is made to pause pruning and restart according to the new schedule. This allows for the case that the new schedule no longer allows pruning at the current time.

sequencer.pruning.set_retention

Summary: Update the pruning retention used by automatic pruning.

Arguments:
- `retention`: com.digitalasset.canton.config.PositiveDurationSeconds

Description: The *retention* is specified as a positive duration and has at most per-second granularity. This call returns an error if no schedule has been configured via *set_schedule* or if automatic pruning has been disabled via *clear_schedule*. Additionally if at the time of this update, pruning is actively running, a best effort is made to pause pruning and restart with the newly specified retention. This allows for the case that the new retention mandates retaining more data than previously.

sequencer.pruning.set_schedule

Summary: Activate automatic pruning according to the specified schedule.

Arguments:
- `cron`: String
- `maxDuration`: com.digitalasset.canton.config.PositiveDurationSeconds
- `retention`: com.digitalasset.canton.config.PositiveDurationSeconds

Description: The schedule is specified in cron format and max_duration and retention durations. The cron string indicates the points in time at which pruning should begin in the GMT time zone, and the maximum duration indicates how long from the start time pruning is allowed to run as long as pruning has not finished pruning up to the specified retention period.

sequencer.pruning.status

Summary: Status of the sequencer and its connected clients

Return type:
- com.digitalasset.canton.domain.sequencing.sequencer.SequencerPruningStatus

Description: Provides a detailed breakdown of information required for pruning: - the current time according to this sequencer instance - domain members that the sequencer supports - for each member when they were registered and whether they are enabled - a list of clients for each member, their last acknowledgement, and whether they are enabled

Health

health.active

Summary: Check if the node is running and is the active instance (mediator, participant)

Return type:
- Boolean

health.dump

Summary: Creates a zip file containing diagnostic information about the canton process running this node

Arguments:
- `outputFile`: better.files.File
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration
- `chunkSize`: Option[Int]

Return type:
- String

health.has_identity

Summary: Returns true if the node has an identity

Return type:
- Boolean

health.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

health.initialized
Summary: Returns true if node has been initialized.
Return type:
- Boolean

health.running
Summary: Check if the node is running
Return type:
- Boolean

health.status
Summary: Get human (and machine) readable status info
Return type:
- com.digitalasset.canton.health.admin.data.NodeStatus[S]

health.wait_for_identity
Summary: Wait for the node to have an identity
Description: This is specifically useful for the Domain Manager which needs its identity to be ready for bootstrapping, but for which we can't rely on wait_for_initialized() because it will be initialized only after being bootstrapped.

health.wait_for_initialized
Summary: Wait for the node to be initialized

health.wait_for_running
Summary: Wait for the node to be running

Database

db.help
Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

db.migrate
Summary: Migrates the instance's database if using a database storage

db.repair_migration
Summary: Only use when advised - repairs the database migration of the instance's database
Arguments:
- `force`: Boolean

Description: In some rare cases, we change already applied database migration files in a new release and the repair command resets the checksums we use to ensure that in general already applied migration files have not been changed. You should only use *db.repair_migration* when advised and otherwise use it at your own risk - in the worst case running it may lead to data corruption when an incompatible database migration (one that

should be rejected because the already applied database migration files have changed) is subsequently falsely applied.

1.28.3.7 Mediator Administration Commands

clear_cache (Testing)
Summary: Clear locally cached variables
Description: Some commands cache values on the client side. Use this command to explicitly clear the caches of these values.

config
Summary: Returns the mediator configuration
Return type:
- com.digitalasset.canton.domain.mediator.MediatorNodeConfig

help
Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

id
Summary: Yields the mediator id of this mediator. Throws an exception, if the id has not yet been allocated (e.g., the mediator has not yet been initialised).
Return type:
- com.digitalasset.canton.topology.MediatorId

is_initialized
Summary: Check if the local instance is running and is fully initialized
Return type:
- Boolean

is_running
Summary: Check if the local instance is running
Return type:
- Boolean

start
Summary: Start the instance

stop
Summary: Stop the instance

Mediator

mediator.clear_schedule
Summary: Deactivate automatic pruning.

mediator.get_schedule
Summary: Inspect the automatic pruning schedule.
Return type:
- Option[com.digitalasset.canton.admin.api.client.data.PruningSchedule]

Description: The schedule consists of a cron expression and max_duration and retention durations. The cron string indicates the points in time at which pruning should begin in the GMT time zone, and the maximum duration indicates how long from the start time pruning is allowed to run as long as pruning has not finished pruning up to the specified retention period. Returns *None* if no schedule has been configured via *set_schedule* or if *clear_schedule* has been invoked.

mediator.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

mediator.initialize

Summary: Initialize a mediator

Arguments:

- `domainId`: com.digitalasset.canton.topology.DomainId
- `mediatorId`: com.digitalasset.canton.topology.MediatorId
- `domainParameters`: com.digitalasset.canton.admin.api.client.data.StaticDomainParameters
- `sequencerConnection`: com.digitalasset.canton.sequencing.SequencerConnection
- `topologySnapshot`: Option[com.digitalasset.canton.topology.store.StoredTopologyTransactions[com.digitalasset.canton.topology.transaction.TopologyChangeOp.Positive]]
- `signingKeyFingerprint`: Option[com.digitalasset.canton.crypto.Fingerprint]

Return type:

- com.digitalasset.canton.crypto.PublicKey

mediator.initialize

Summary: Initialize a mediator

Arguments:

- `domainId`: com.digitalasset.canton.topology.DomainId
- `mediatorId`: com.digitalasset.canton.topology.MediatorId
- `domainParameters`: com.digitalasset.canton.admin.api.client.data.StaticDomainParameters
- `sequencerConnections`: com.digitalasset.canton.sequencing.SequencerConnections
- `topologySnapshot`: Option[com.digitalasset.canton.topology.store.StoredTopologyTransactions[com.digitalasset.canton.topology.transaction.TopologyChangeOp.Positive]]
- `signingKeyFingerprint`: Option[com.digitalasset.canton.crypto.Fingerprint]

Return type:

- com.digitalasset.canton.crypto.PublicKey

mediator.locate_pruning_timestamp

Summary: Obtain a timestamp at or near the beginning of mediator state

Arguments:

- `index`: com.digitalasset.canton.config.RequireTypes.PositiveInt

Return type:

- Option[com.digitalasset.canton.data.CantonTimestamp]

Description: This command provides insight into the current state of mediator pruning when called with the default value of *index* 1. When pruning the mediator manually via

prune_at and with the intent to prune in batches, specify a value such as 1000 to obtain a pruning timestamp that corresponds to the end of the batch.

mediator.prune

Summary: Prune the mediator of unnecessary data while keeping data for the default retention period

Description: Removes unnecessary data from the Mediator that is earlier than the default retention period. The default retention period is set in the configuration of the canton node running this command under *parameters.retention-period-defaults.mediator*.

mediator.prune_at

Summary: Prune the mediator of unnecessary data up to and including the given timestamp

Arguments:

- `timestamp`: com.digitalasset.canton.data.CantonTimestamp

mediator.prune_with_retention_period

Summary: Prune the mediator of unnecessary data while keeping data for the provided retention period

Arguments:

- `retentionPeriod`: scala.concurrent.duration.FiniteDuration

mediator.set_cron

Summary: Modify the cron used by automatic pruning.

Arguments:

- `cron`: String

Description: The schedule is specified in cron format and refers to pruning start times in the GMT time zone. This call returns an error if no schedule has been configured via *set_schedule* or if automatic pruning has been disabled via *clear_schedule*. Additionally if at the time of this modification, pruning is actively running, a best effort is made to pause pruning and restart according to the new schedule. This allows for the case that the new schedule no longer allows pruning at the current time.

mediator.set_max_duration

Summary: Modify the maximum duration used by automatic pruning.

Arguments:

- `maxDuration`: com.digitalasset.canton.config.PositiveDurationSeconds

Description: The *maxDuration* is specified as a positive duration and has at most per-second granularity. This call returns an error if no schedule has been configured via *set_schedule* or if automatic pruning has been disabled via *clear_schedule*. Additionally if at the time of this modification, pruning is actively running, a best effort is made to pause pruning and restart according to the new schedule. This allows for the case that the new schedule no longer allows pruning at the current time.

mediator.set_retention

Summary: Update the pruning retention used by automatic pruning.

Arguments:

- `retention`: com.digitalasset.canton.config.PositiveDurationSeconds

Description: The *retention* is specified as a positive duration and has at most per-second granularity. This call returns an error if no schedule has been configured via *set_schedule* or if automatic pruning has been disabled via *clear_schedule*. Additionally if at the time of this update, pruning is actively running, a best effort is made to pause pruning and restart with the newly specified retention. This allows for the case that the new retention mandates retaining more data than previously.

mediator.set_schedule

Summary: Activate automatic pruning according to the specified schedule.
Arguments:
- `cron`: String
- `maxDuration`: com.digitalasset.canton.config.PositiveDurationSeconds
- `retention`: com.digitalasset.canton.config.PositiveDurationSeconds

Description: The schedule is specified in cron format and max_duration and retention durations. The cron string indicates the points in time at which pruning should begin in the GMT time zone, and the maximum duration indicates how long from the start time pruning is allowed to run as long as pruning has not finished pruning up to the specified retention period.

Health

health.active

Summary: Check if the node is running and is the active instance (mediator, participant)
Return type:
- Boolean

health.dump

Summary: Creates a zip file containing diagnostic information about the canton process running this node
Arguments:
- `outputFile`: better.files.File
- `timeout`: com.digitalasset.canton.config.NonNegativeDuration
- `chunkSize`: Option[Int]

Return type:
- String

health.has_identity

Summary: Returns true if the node has an identity
Return type:
- Boolean

health.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

health.initialized

Summary: Returns true if node has been initialized.
Return type:
- Boolean

health.running

Summary: Check if the node is running
Return type:
- Boolean

health.status

Summary: Get human (and machine) readable status info
Return type:

- com.digitalasset.canton.health.admin.data.NodeStatus[S]

health.wait_for_identity

Summary: Wait for the node to have an identity
Description: This is specifically useful for the Domain Manager which needs its identity to be ready for bootstrapping, but for which we can't rely on wait_for_initialized() because it will be initialized only after being bootstrapped.

health.wait_for_initialized

Summary: Wait for the node to be initialized

health.wait_for_running

Summary: Wait for the node to be running

Database

db.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

db.migrate

Summary: Migrates the instance's database if using a database storage

db.repair_migration

Summary: Only use when advised - repairs the database migration of the instance's database
Arguments:
- `force`: Boolean

Description: In some rare cases, we change already applied database migration files in a new release and the repair command resets the checksums we use to ensure that in general already applied migration files have not been changed. You should only use *db.repair_migration* when advised and otherwise use it at your own risk - in the worst case running it may lead to data corruption when an incompatible database migration (one that should be rejected because the already applied database migration files have changed) is subsequently falsely applied.

Sequencer Connection

sequencer_connection.get

Summary: Get Sequencer Connection
Return type:
- Option[com.digitalasset.canton.sequencing.SequencerConnections]

Description: Use this command to get the currently configured sequencer connection details for this sequencer client. If this node has not yet been initialized, this will return None.

sequencer_connection.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:

- `methodName`: String

sequencer_connection.modify

Summary: Modify Default Sequencer Connection

Arguments:

- `modifier`: com.digitalasset.canton.sequencing.SequencerConnection => com.digitalasset.canton.sequencing.SequencerConnection

Description: Modify sequencer connection details for this sequencer client node, by passing a modifier function that operates on the existing default connection.

sequencer_connection.modify_connections

Summary: Modify Sequencer Connections

Arguments:

- `modifier`: com.digitalasset.canton.sequencing.SequencerConnections => com.digitalasset.canton.sequencing.SequencerConnections

Description: Modify sequencer connection details for this sequencer client node, by passing a modifier function that operates on the existing connection configuration.

sequencer_connection.set

Summary: Set Sequencer Connection

Arguments:

- `connection`: com.digitalasset.canton.sequencing.SequencerConnection

Description: Set new sequencer connection details for this sequencer client node. This will replace any pre-configured connection details. This command will only work after the node has been initialized.

sequencer_connection.set

Summary: Set Sequencer Connection

Arguments:

- `connections`: com.digitalasset.canton.sequencing.SequencerConnections

Description: Set new sequencer connection details for this sequencer client node. This will replace any pre-configured connection details. This command will only work after the node has been initialized.

1.29 Monitoring

1.29.1 Introduction

Observability (also known as monitoring) lets you determine if the Daml Enterprise solution is healthy or not. If the state is not healthy, observability helps diagnose the root cause. There are three parts to observability: metrics, logs, and traces. These are described in this section.

To avoid becoming overwhelmed by the number of metrics and log messages, follow these steps:

> Read the shortcut to learning what is important, which is described below in the section *Hands-On with the Daml Enterprise - Observability Example* as a starting point and inspiration when building your metric monitoring.
> For an overview of how most metrics are exposed, read the section *Golden Signals and Key Metrics Quick Start* below. It describes the philosophy behind metric naming and labeling.

The remaining sections provide references to more detailed information.

1.29.1.1 Hands-On with the Daml Enterprise - Observability Example

The Daml Enterprise - Observability Example GitHub repository provides a complete reference example for exploring the metrics that Daml Enterprise exposes. You can use it to explore the collection, aggregation, filtering, and visualization of metrics. It is self-contained, with the following components:

An example Docker compose file to create a run-time for all the components
Some shell scripts to generate requests to the Daml Enterprise solution
A Prometheus config file to scrape the metrics data
A Grafana template file(s) to visualize the metrics in a meaningful way, such as shown below in the example dashboard

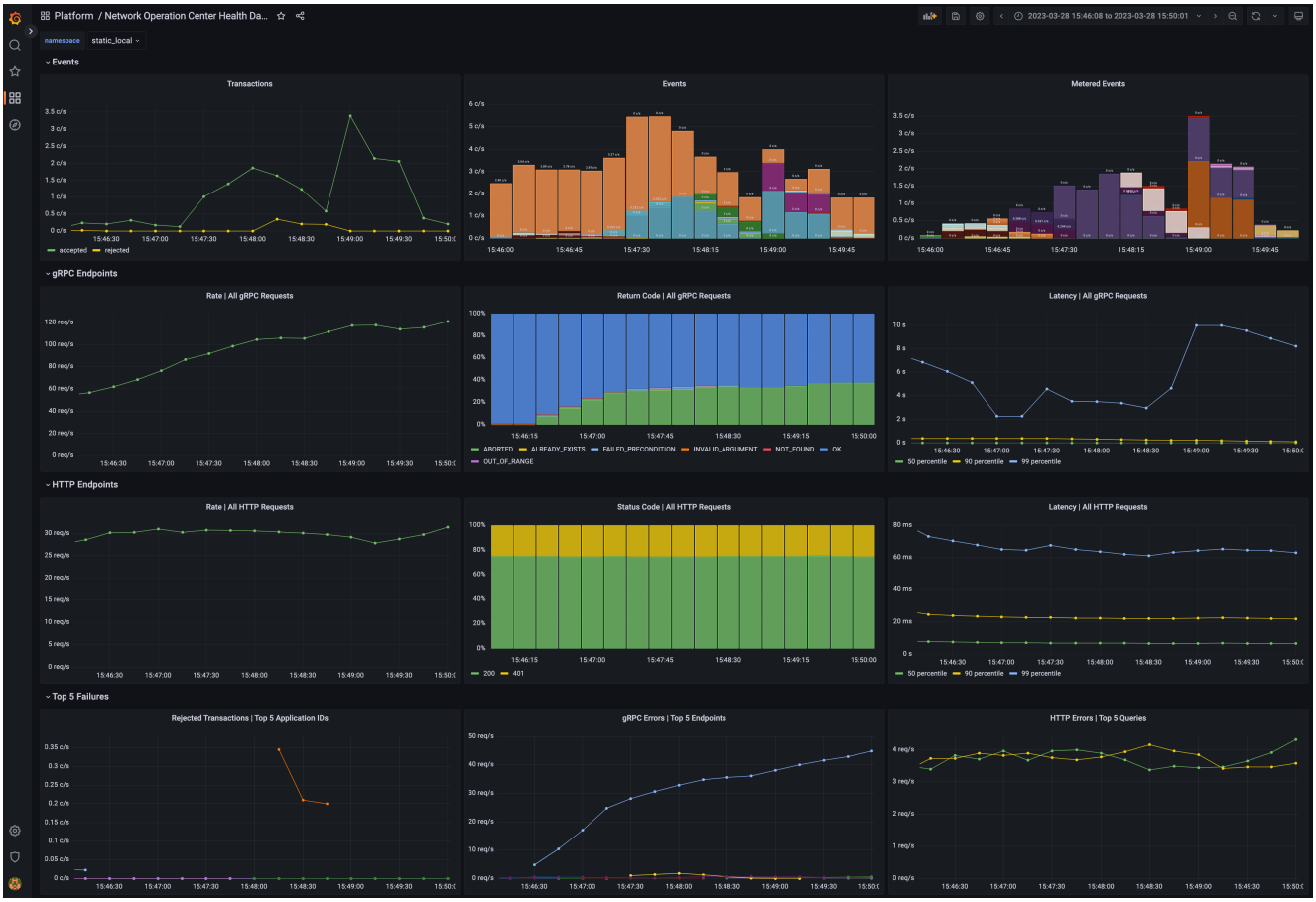


Fig. 17: Dashboard with metrics

1.29.2 Golden Signals and Key Metrics Quick Start

The best practice for monitoring a microservices application is an approach known as the Golden Signals, or the RED method. In this approach, metric monitoring determines whether the application is healthy and, if not healthy, which service is the root cause of the issue. The Golden Signals for HTTP and gRPC endpoints are supported for all endpoints. Key metrics specific to Daml Enterprises are also available. These are described below.

The following Golden Signal metrics for each HTTP and gRPC API are available:

Input request rate, as a counter
Error rate, as a counter (discussed below)
Latency (the time to process a request), as a histogram
Size of the payload, as a counter, following the Apache HTTP precedent

You can filter or aggregate each metric using its accompanying labels. The instrumentation labels added to each HTTP API metric are as follows:

`http_verb`: the HTTP verb (for example: GET, POST)
`http_status`: the status code (for example: 200, 401, 403, 504)
`host`: the host identifier
`daml_version`: the Daml release number
`service`: a string to identify what Daml service or Canton component is running in this process (for example: `participant`, `domain`, `json_api`), as well as `domain` if several Canton components run in a single process
`path`: the request made to the endpoint (for example: `/v1/create`, `/v1/exercise`)

The gRPC protocol is layered on top of HTTP/2, so certain labels (such as the `daml_version` and `service`) from the above section are included. The labels added by default to each gRPC API metric are as follows:

`canton_version`: the Canton protocol version
`grpc_code`: the human readable status code for gRPC (for example: `OK`, `CANCELLED`, `DEADLINE_EXCEEDED`)
The type of the client/server gRPC request, under the labels `grpc_client_type` and `grpc_server_type`
The protobuf package and service names, under the labels `grpc_service_name` and `grpc_method_name`

The following other key metrics are monitored:

A binary gauge indicates whether the node is healthy or not healthy. This can also be used to infer which node is passive in a highly available configuration because it will show as not being healthy, while the active node is always healthy.
A binary gauge signals whether a node is active or passive, for identifying which node is the active node.
A binary gauge detects when pruning is occurring.
Each participant node measures the count of the inflight (dirty) requests so the user can see if `maxDirtyRequests` limit is close to being hit. The metrics are: `canton_dirty_requests` and `canton_max_dirty_requests`.
Each participant node records the distribution of events (updates) received by the participant and allows drill-down by event type (package upload, party creation, or transaction), status (success or failure), participant ID, and application ID (if available). The counter is called `daml_indexer_events_total`.

The ledger event requests are totaled in a counter called `daml_indexer_metered_events_total`.
Metrics are available for monitoring the usage of JVM execution services used by Daml components.
JVM garbage collection metrics are collected.

This list is not exhaustive. It highlights the most important metrics.

1.29.3 Set Up Metrics Scraping

1.29.3.1 Enable the Prometheus Reporter

Prometheus is recommended for metrics reporting. Other reporters (jmx, graphite, and csv) are supported, but they are deprecated. Any such reporter should be migrated to Prometheus.

Prometheus can be enabled using:

```
canton.monitoring.metrics.reporters = [{
  type = prometheus
  address = "localhost" // default
  port = 9000 // default
}]
```

1.29.3.2 Prometheus-Only Metrics

Some metrics are available only when using the Prometheus reporter. These metrics include common gRPC and HTTP metrics (which help you to measure the four golden signals), Java Executor Services metrics, and JVM GC and memory usage metrics (if enabled). The metrics are documented in detail here.

Any metric marked with * is available only when using the Prometheus reporter.

1.29.3.3 Deprecated Reporters

JMX-based reporting (for testing purposes only) can be enabled using:

```
canton.monitoring.metrics.reporters = [{ type = jmx }]
```

Additionally, metrics can be written to a file:

```
canton.monitoring.metrics.reporters = [{
  type = jmx
}, {
  type = csv
  directory = "metrics"
  interval = 5s // default
  filters = [{
    contains = "canton"
  }]
}]
```

or reported via Graphite (to Grafana) using:

```
canton.monitoring.metrics.reporters = [{
  type = graphite
  address = "localhost" // default
  port = 2003
  prefix.type = hostname // default
  interval = 30s // default
  filters = [{
    contains = "canton"
  }]
}]
```

When using the `graphite` or the `csv` reporter, Canton periodically evaluates all metrics matching the given filters. Filter for only those metrics that are relevant to you.

In addition to Canton metrics, the process can also report Daml metrics (of the Ledger API server). Optionally, JVM metrics can be included using:

```
canton.monitoring.metrics.report-jvm-metrics = yes // default no
```

1.29.4 Metrics

The following sections contain the common metrics exposed for Daml services supporting a Prometheus metrics reporter.

For the metric types referenced below, see the relevant Prometheus documentation.

1.29.4.1 Participant Metrics

canton.<domain>.conflict-detection.sequencer-counter-queue

Summary: Size of conflict detection sequencer counter queue
Description: The task scheduler will work off tasks according to the timestamp order, scheduling the tasks whenever a new timestamp has been observed. This metric exposes the number of un-processed sequencer messages that will trigger a timestamp advancement.
Type: Counter
Qualification: Debug

canton.<domain>.conflict-detection.task-queue

Summary: Size of conflict detection task queue
Description: The task scheduler will schedule tasks to run at a given timestamp. This metric exposes the number of tasks that are waiting in the task queue for the right time to pass. A huge number does not necessarily indicate a bottleneck; it could also mean that a huge number of tasks have not yet arrived at their execution time.
Type: Gauge
Qualification: Debug

canton.<domain>.dirty-requests

Summary: Size of conflict detection task queue
Description: The task scheduler will schedule tasks to run at a given timestamp. This metric exposes the number of tasks that are waiting in the task queue for the right time to pass. A huge number does not necessarily indicate a bottleneck; it could also mean that a huge number of tasks have not yet arrived at their execution time.
Type: Counter
Qualification: Debug

canton.<domain>.protocol-messages.confirmation-request-creation

Summary: Time to create a confirmation request
Description: The time that the transaction protocol processor needs to create a confirmation request.
Type: Timer
Qualification: Debug

canton.<domain>.protocol-messages.confirmation-request-size

Summary: Confirmation request size
Description: Records the histogram of the sizes of (transaction) confirmation requests.
Type: Histogram
Qualification: Debug

canton.<domain>.protocol-messages.transaction-message-receipt

Summary: Time to parse a transaction message
Description: The time that the transaction protocol processor needs to parse and decrypt an incoming confirmation request.
Type: Timer
Qualification: Debug

canton.<domain>.request-tracker.sequencer-counter-queue

Summary: Size of record order publisher sequencer counter queue
Description: Same as for conflict-detection, but measuring the sequencer counter queues for the publishing to the ledger api server according to record time.
Type: Counter
Qualification: Debug

canton.<domain>.request-tracker.task-queue

Summary: Size of record order publisher task queue
Description: The task scheduler will schedule tasks to run at a given timestamp. This metric exposes the number of tasks that are waiting in the task queue for the right time to pass.
Type: Gauge
Qualification: Debug

canton.<domain>.sequencer-client.application-handle

Summary: Timer monitoring time and rate of sequentially handling the event application logic
Description: All events are received sequentially. This handler records the the rate and time it takes the application (participant or domain) to handle the events.
Type: Timer
Qualification: Debug

canton.<domain>.sequencer-client.delay

Summary: The delay on the event processing
Description: Every message received from the sequencer carries a timestamp that was assigned by the sequencer when it sequenced the message. This timestamp is called the sequencing timestamp. The component receiving the message on the participant, mediator or topology manager side, is the sequencer client. Upon receiving the message, the sequencer client compares the time difference between the sequencing time and the computers local clock and exposes this difference as the given metric. The difference will include the clock-skew and the processing latency between assigning the timestamp on the sequencer and receiving the message by the recipient. If the difference is large compared to the usual latencies and if clock skew can be ruled out, then it means that the node is still trying to catch up with events that were sequenced by the sequencer a while ago. This can happen after having been offline for a while or if the node is too slow to keep up with the messaging load.
Type: Gauge
Qualification: Debug

canton.<domain>.sequencer-client.event-handle

Summary: Timer monitoring time and rate of entire event handling
Description: Most event handling cost should come from the application-handle. This timer measures the full time (which should just be marginally more than the application handle.
Type: Timer
Qualification: Debug

canton.<domain>.sequencer-client.handler.actual-in-flight-event-batches

Summary: Nodes process the events from the domain's sequencer in batches. This metric tracks how many such batches are processed in parallel.
Description: Incoming messages are processed by a sequencer client, which combines them into batches of size up to 'event-inbox-size' before sending them to an application handler for processing. Depending on the system's configuration, the rate at which event batches are sent to the handler may be throttled to avoid overwhelming it with too many events at once. Indicators that the configured upper bound may be too low: This metric constantly is closed to the configured maximum, which is exposed via 'max-in-flight-event-batches', while the system's resources are under-utilized. Indicators that the configured upper bound may be too high: Out-of-memory errors crashing the JVM or frequent garbage collection cycles that slow down processing. The metric tracks how many of these batches have been sent to the application handler but have not yet been fully processed. This metric can help identify potential bottlenecks or issues with the application's processing of events and provide insights into the overall workload of the system.
Type: Counter
Qualification: Saturation

canton.<domain>.sequencer-client.handler.max-in-flight-event-batches

Summary: Nodes process the events from the domain's sequencer in batches. This metric tracks the upper bound of such batches being processed in parallel.
Description: Incoming messages are processed by a sequencer client, which combines them into batches of size up to 'event-inbox-size' before sending them to an application handler for processing. Depending on the system's configuration, the rate at which event batches are sent to the handler may be throttled to avoid overwhelming it with too many events at once. Configured by 'maximum-in-flight-event-batches' parameter in the sequencer-client config The metric shows the configured upper limit on how many batches the application handler may process concurrently. The metric 'actual-in-flight-event-batches' tracks the actual number of currently processed batches.
Type: Gauge
Qualification: Saturation

canton.<domain>.sequencer-client.load

Summary: The load on the event subscription
Description: The event subscription processor is a sequential process. The load is a factor between 0 and 1 describing how much of an existing interval has been spent in the event handler.
Type: Gauge
Qualification: Debug

canton.<domain>.sequencer-client.submissions.dropped

Summary: Count of send requests that did not cause an event to be sequenced
Description: Counter of send requests we did not witness a corresponding event to be sequenced by the supplied max-sequencing-time. There could be many reasons for this happening: the request may have been lost before reaching the sequencer, the sequencer may be at capacity and the the max-sequencing-time was exceeded by the time the request was processed, or the supplied max-sequencing-time may just be too small for the sequencer to be able to sequence the request.
Type: Counter
Qualification: Debug

canton.<domain>.sequencer-client.submissions.in-flight

Summary: Number of sequencer send requests we have that are waiting for an outcome or timeout
Description: Incremented on every successful send to the sequencer. Decremented when the event or an error is sequenced, or when the max-sequencing-time has elapsed.
Type: Counter
Qualification: Debug

canton.<domain>.sequencer-client.submissions.overloaded

Summary: Count of send requests which receive an overloaded response
Description: Counter that is incremented if a send request receives an overloaded response from the sequencer.
Type: Counter
Qualification: Debug

canton.<domain>.sequencer-client.submissions.sends

Summary: Rate and timings of send requests to the sequencer
Description: Provides a rate and time of how long it takes for send requests to be accepted by the sequencer. Note that this is just for the request to be made and not for the requested event to actually be sequenced.
Type: Timer
Qualification: Debug

canton.<domain>.sequencer-client.submissions.sequencing

Summary: Rate and timings of sequencing requests
Description: This timer is started when a submission is made to the sequencer and then completed when a corresponding event is witnessed from the sequencer, so will encompass the entire duration for the sequencer to sequence the request. If the request does not result in an event no timing will be recorded.
Type: Timer
Qualification: Debug

canton.commitments.compute

Summary: Time spent on commitment computations.
Description: Participant nodes compute bilateral commitments at regular intervals. This metric exposes the time spent on each computation. If the time to compute the metrics starts to exceed the commitment intervals, this likely indicates a problem.
Type: Timer
Qualification: Debug

canton.db-storage.<service>.executor.queued

Summary: Number of database access tasks waiting in queue
Description: Database access tasks get scheduled in this queue and get executed using one of the existing asynchronous sessions. A large queue indicates that the database connection is not able to deal with the large number of requests. Note that the queue has a maximum size. Tasks that do not fit into the queue will be retried, but won't show up in this metric.
Type: Counter
Qualification: Debug
Instances: locks, write, general

canton.db-storage.<service>.executor.running

Summary: Number of database access tasks currently running
Description: Database access tasks run on an async executor. This metric shows the current number of tasks running in parallel.
Type: Counter
Qualification: Debug
Instances: locks, write, general

canton.db-storage.<service>.executor.waittime

Summary: Scheduling time metric for database tasks
Description: Every database query is scheduled using an asynchronous executor with a queue. The time a task is waiting in this queue is monitored using this metric.
Type: Timer
Qualification: Debug
Instances: locks, write, general

canton.db-storage.<storage>

Summary: Timer monitoring duration and rate of accessing the given storage
Description: Covers both read from and writes to the storage.
Type: Timer
Qualification: Debug

canton.db-storage.<storage>.load

Summary: The load on the given storage
Description: The load is a factor between 0 and 1 describing how much of an existing interval has been spent reading from or writing to the storage.
Type: Gauge
Qualification: Debug

canton.db-storage.alerts.multi-domain-event-log

Summary: Number of failed writes to the multi-domain event log
Description: Failed writes to the multi domain event log indicate an issue requiring user intervention. In the case of domain event logs, the corresponding domain no longer emits any subsequent events until domain recovery is initiated (e.g. by disconnecting and reconnecting the participant from the domain). In the case of the participant event log, an operation might need to be reissued. If this counter is larger than zero, check the canton log for errors for details.
Type: Counter
Qualification: Debug

canton.db-storage.alerts.single-dimension-event-log

Summary: Number of failed writes to the event log
Description: Failed writes to the single dimension event log indicate an issue requiring user intervention. In the case of domain event logs, the corresponding domain no longer emits any subsequent events until domain recovery is initiated (e.g. by disconnecting and reconnecting the participant from the domain). In the case of the participant event log, an operation might need to be reissued. If this counter is larger than zero, check the canton log for errors for details.
Type: Counter
Qualification: Debug

canton.dirty_requests*

Summary: Number of requests being validated.
Description: Number of requests that are currently being validated. This also covers requests submitted by other participants.
Type: Gauge
Qualification: Debug
Labels:

- **participant**: The id of the participant for which the value applies.

canton.max_dirty_requests*

Summary: Configured maximum number of requests currently being validated.
Description: Configuration for the maximum number of requests that are currently being validated. This also covers requests submitted by other participants. A negative value means no configuration value was provided and no limit is enforced.
Type: Gauge
Qualification: Debug
Labels:
- **participant**: The id of the participant for which the value applies.

canton.prune

Summary: Duration of prune operations.
Description: This timer exposes the duration of pruning requests from the Canton portion of the ledger.
Type: Timer
Qualification: Debug

canton.prune.max-event-age

Summary: Age of oldest unpruned event.
Description: This gauge exposes the age of the oldest, unpruned event in hours as a way to quantify the pruning backlog.
Type: Gauge
Qualification: Debug

canton.updates-published

Summary: Number of updates published through the read service to the indexer
Description: When an update is published through the read service, it has already been committed to the ledger. The indexer will subsequently store the update in a form that allows for querying the ledger efficiently.
Type: Meter
Qualification: Debug

daml.cache.evicted_weight*

Summary: The sum of weights of cache entries evicted.
Description: The total weight of the entries evicted from the cache.
Type: Counter
Qualification: Debug
Labels:
- **name**: The cache for which the metrics are registered.

daml.cache.evictions*

Summary: The number of the evicted cache entries.
Description: When an entry is evicted from the cache, the counter is incremented.
Type: Counter
Qualification: Debug
Labels:
- **name**: The cache for which the metrics are registered.

daml.cache.hits*

Summary: The number of cache hits.
Description: When a cache lookup encounters an existing cache entry, the counter is incremented.
Type: Counter
Qualification: Debug
Labels:
- **name**: The cache for which the metrics are registered.

daml.cache.misses*

Summary: The number of cache misses.
Description: When a cache lookup first encounters a missing cache entry, the counter is incremented.
Type: Counter
Qualification: Debug
Labels:
- **name**: The cache for which the metrics are registered.

daml.commands.delayed_submissions

Summary: The number of the delayed Daml commands.
Description: The number of Daml commands that have been delayed internally because they have been evaluated to require the ledger time further in the future than the expected latency.
Type: Meter
Qualification: Debug

daml.commands.failed_command_interpretations

Summary: The number of Daml commands that failed in interpretation.
Description: The number of Daml commands that have been rejected by the interpreter (e.g. badly authorized action).
Type: Meter
Qualification: Debug

daml.commands.max_in_flight_capacity

Summary: The maximum number of Daml commands that can await completion.
Description: The maximum number of Daml commands that can await completion in the Command Service.
Type: Counter
Qualification: Debug

daml.commands.max_in_flight_length

Summary: The number of the Daml commands awaiting completion.
Description: The number of the currently Daml commands awaiting completion in the Command Service.
Type: Counter
Qualification: Debug

daml.commands.reassignment_validation

Summary: The time to validate a reassignment command.
Description: The time to validate a submitted Daml command before is fed to the interpreter.
Type: Timer
Qualification: Debug

daml.commands.submissions

Summary: The time to fully process a Daml command.
Description: The time to validate and interpret a command before it is handed over to the synchronization services to be finalized (either committed or rejected).
Type: Timer
Qualification: Latency

daml.commands.submissions_running

Summary: The number of the Daml commands that are currently being handled by the ledger api server.
Description: The number of the Daml commands that are currently being handled by the ledger api server (including validation, interpretation, and handing the transaction over to the synchronization services).
Type: Counter
Qualification: Debug

daml.commands.valid_submissions

Summary: The total number of the valid Daml commands.
Description: The total number of the Daml commands that have passed validation and were sent to interpretation in this ledger api server process.
Type: Meter
Qualification: Debug

daml.commands.validation

Summary: The time to validate a Daml command.
Description: The time to validate a submitted Daml command before is fed to the interpreter.
Type: Timer
Qualification: Debug

daml.db.commit.duration.seconds*

Summary: The time needed to perform the SQL query commit.
Description: This metric measures the time it takes to commit an SQL transaction relating to the <operation>. It roughly corresponds to calling *commit()* on a DB connection.
Type: Timer
Qualification: Debug
Labels:
- **name**: The operation/pool for which the metric is registered.

daml.db.compression.duration.seconds*

Summary: The time needed to decompress the SQL query result.
Description: Some index database queries that target contracts involve a decompression step. For such queries this metric represents the time it takes to decompress contract arguments retrieved from the database.
Type: Timer
Qualification: Debug
Labels:
- **name**: The operation/pool for which the metric is registered.

daml.db.exec.duration.seconds*

Summary: The time needed to run the SQL query and read the result.
Description: This metric encompasses the time measured by *query* and *commit* metrics. Additionally it includes the time needed to obtain the DB connection, optionally roll it back and close the connection at the end.
Type: Timer
Qualification: Debug
Labels:
- **name**: The operation/pool for which the metric is registered.

daml.db.query.duration.seconds*

Summary: The time needed to run the SQL query.
Description: This metric measures the time it takes to execute a block of code (on a dedicated executor) related to the <operation> that can issue multiple SQL statements such that all run in a single DB transaction (either committed or aborted).
Type: Timer
Qualification: Debug
Labels:
- **name**: The operation/pool for which the metric is registered.

daml.db.translation.duration.seconds*

Summary: The time needed to turn serialized Daml-LF values into in-memory objects.
Description: Some index database queries that target contracts and transactions involve a Daml-LF translation step. For such queries this metric stands for the time it takes to turn the serialized Daml-LF values into in-memory representation.
Type: Timer
Qualification: Debug
Labels:
- **name**: The operation/pool for which the metric is registered.

daml.db.wait.duration.seconds*

Summary: The time needed to acquire a connection to the database.
Description: SQL statements are run in a dedicated executor. This metric measures the time it takes between creating the SQL statement corresponding to the <operation> and the point when it starts running on the dedicated executor.
Type: Timer
Qualification: Debug
Labels:
- **name**: The operation/pool for which the metric is registered.

daml.execution.cache.contract_state.register_update

Summary: The time spent to update the cache.
Description: The total time spent in sequential update steps of the contract state caches updating logic. This metric is created with debugging purposes in mind.
Type: Timer
Qualification: Debug

daml.execution.cache.key_state.register_update

Summary: The time spent to update the cache.
Description: The total time spent in sequential update steps of the contract state caches updating logic. This metric is created with debugging purposes in mind.
Type: Timer
Qualification: Debug

daml.execution.cache.read_through_not_found

Summary: The number of cache read-throughs resulting in not found contracts.
Description: On cache misses, a read-through query is performed against the Index database. When the contract is not found (as result of this query), this counter is incrmented.
Type: Counter
Qualification: Debug

daml.execution.cache.resolve_divulgence_lookup

Summary: The number of lookups trying to resolve divulged contracts on active contracts cache hits.
Description: Divulged contracts are not cached in the contract state caches. On active contract cache hits, where stakeholders are not within the submission readers, a contract activeness lookup is performed against the Index database. On such lookups, this counter is incremented.
Type: Counter
Qualification: Debug

daml.execution.cache.resolve_full_lookup

Summary: The number of lookups trying to resolve divulged contracts on archived contracts cache hits.
Description: Divulged contracts are not cached in the contract state caches. On archived contract cache hits, where stakeholders are not within the submission readers, a full contract activeness lookup (including fetching contract arguments) is performed against the Index database. On such lookups, this counter is incremented.
Type: Counter
Qualification: Debug

daml.execution.engine

Summary: The time spent executing a Daml command.
Description: The time spent by the Daml engine executing a Daml command (excluding fetching data).
Type: Timer
Qualification: Debug

daml.execution.engine_running

Summary: The number of Daml commands currently being executed.
Description: The number of the commands that are currently being executed by the Daml engine (excluding fetching data).
Type: Counter
Qualification: Debug

daml.execution.get_lf_package

Summary: The time to fetch individual Daml code packages during interpretation.
Description: The interpretation of a command in the ledger api server might require fetching multiple Daml packages. This metric exposes the time needed to fetch the packages that are necessary for interpretation.
Type: Timer
Qualification: Debug

daml.execution.lookup_active_contract

Summary: The time to lookup individual active contracts during interpretation.
Description: The interpretation of a command in the ledger api server might require fetching multiple active contracts. This metric exposes the time to lookup individual active contracts.
Type: Timer
Qualification: Debug

daml.execution.lookup_active_contract_count_per_execution

Summary: The number of the active contracts looked up per Daml command.
Description: The interpretation of a command in the ledger api server might require fetching multiple active contracts. This metric exposes the number of active contracts that must be looked up to process a Daml command.
Type: Histogram
Qualification: Debug

daml.execution.lookup_active_contract_per_execution

Summary: The compound time to lookup all active contracts in a single Daml command.
Description: The interpretation of a command in the ledger api server might require fetching multiple active contracts. This metric exposes the compound time to lookup all the active contracts in a single Daml command.
Type: Timer
Qualification: Debug

daml.execution.lookup_contract_key

Summary: The time to lookup individual contract keys during interpretation.
Description: The interpretation of a command in the ledger api server might require fetching multiple contract keys. This metric exposes the time needed to lookup individual contract keys.
Type: Timer
Qualification: Debug

daml.execution.lookup_contract_key_count_per_execution

Summary: The number of contract keys looked up per Daml command.
Description: The interpretation of a command in the ledger api server might require fetching multiple contract keys. This metric exposes the number of contract keys that must be looked up to process a Daml command.
Type: Histogram
Qualification: Debug

daml.execution.lookup_contract_key_per_execution

Summary: The compound time to lookup all contract keys in a single Daml command.
Description: The interpretation of a command in the ledger api server might require fetching multiple contract keys. This metric exposes the compound time needed to lookup all the contract keys in a single Daml command.
Type: Timer
Qualification: Debug

daml.execution.retry

Summary: The number of the interpretation retries.
Description: The total number of interpretation retries attempted due to mismatching ledger effective time in this ledger api server process.
Type: Meter
Qualification: Debug

daml.execution.total

Summary: The overall time spent interpreting a Daml command.
Description: The time spent interpreting a Daml command in the ledger api server (includes executing Daml and fetching data).
Type: Timer
Qualification: Debug

daml.execution.total_running

Summary: The number of Daml commands currently being interpreted.
Description: The number of the commands that are currently being interpreted (includes executing Daml code and fetching data).
Type: Counter
Qualification: Debug

daml.executor.runtime.completed*

Summary: The number of tasks completed in an instrumented executor.
Description: The number of tasks completed by this executor
Type: Meter
Qualification: Debug
Labels:
- **name**: The name of the executor service.
- **type**: The type of the executor service. Can be *fork_join* or *thread_pool*.

daml.executor.runtime.duration.seconds*

Summary: The time a task runs in an instrumented executor.
Description: A task is considered running only after it has started execution.
Type: Timer
Qualification: Debug
Labels:
- **name**: The name of the executor service.
- **type**: The type of the executor service. Can be *fork_join* or *thread_pool*.

daml.executor.runtime.idle.duration.seconds*

Summary: The time that a task is idle in an instrumented executor.
Description: A task is considered idle if it was submitted to the executor but it has not started execution yet.
Type: Timer
Qualification: Debug
Labels:
- **name**: The name of the executor service.
- **type**: The type of the executor service. Can be *fork_join* or *thread_pool*.

daml.executor.runtime.running*

Summary: The number of tasks running in an instrumented executor.
Description: The number of currently running tasks.
Type: Counter
Qualification: Debug
Labels:
- **name**: The name of the executor service.
- **type**: The type of the executor service. Can be *fork_join* or *thread_pool*.

daml.executor.runtime.submitted*

Summary: The number of tasks submitted to an instrumented executor.
Description: Number of tasks that were submitted to the executor.
Type: Meter
Qualification: Debug
Labels:
- **name**: The name of the executor service.
- **type**: The type of the executor service: *fork_join* or *thread_pool*.

daml.index.active_contracts_buffer_size

Summary: The buffer size for active contracts requests.
Description: An Akka stream buffer is added at the end of all streaming queries, allowing to absorb temporary downstream backpressure (e.g. when the client is slower than upstream delivery throughput). This metric gauges the size of the buffer for queries requesting active contracts that transactions satisfying a given predicate.
Type: Counter
Qualification: Saturation

daml.index.completions_buffer_size

Summary: The buffer size for completions requests.
Description: An Akka stream buffer is added at the end of all streaming queries, allowing to absorb temporary downstream backpressure (e.g. when the client is slower than upstream delivery throughput). This metric gauges the size of the buffer for queries requesting the completed commands in a specific period of time.
Type: Counter
Qualification: Saturation

daml.index.db.active_contract_lookup_batch_size

Summary: The batch sizes in the active contract lookup batch-loading Contract Service.
Description: The number of active contract lookups contained in a batch, used in the batch-loading Contract Service.
Type: Histogram
Qualification: Debug

daml.index.db.active_contract_lookup_buffer_capacity

Summary: The capacity of the active contract lookup queue.
Description: The maximum number of elements that can be kept in the queue of active contract lookups in the batch-loading queue of the Contract Service.
Type: Counter
Qualification: Debug

daml.index.db.active_contract_lookup_buffer_delay

Summary: The queuing delay for the active contract lookup queue.
Description: The queuing delay for the pending active contract lookups in the batch-loading queue of the Contract Service.
Type: Timer
Qualification: Debug

daml.index.db.active_contract_lookup_buffer_length

Summary: The number of the currently pending active contract lookups.
Description: The number of the currently pending active contract lookups in the batch-loading queue of the Contract Service.
Type: Counter
Qualification: Debug

daml.index.db.compression.create_argument_compressed

Summary: The size of the compressed arguments of a create event.
Description: Event information can be compressed by the indexer before storing it in the database. This metric collects statistics about the size of compressed arguments of a create event.
Type: Histogram
Qualification: Debug

daml.index.db.compression.create_argument_uncompressed

Summary: The size of the decompressed argument of a create event.
Description: Event information can be compressed by the indexer before storing it in the database. This metric collects statistics about the size of decompressed arguments of a create event.
Type: Histogram
Qualification: Debug

daml.index.db.compression.create_key_value_compressed

Summary: The size of the compressed key value of a create event.
Description: Event information can be compressed by the indexer before storing it in the database. This metric collects statistics about the size of compressed key value of a create event.
Type: Histogram
Qualification: Debug

daml.index.db.compression.create_key_value_uncompressed

Summary: The size of the decompressed key value of a create event.
Description: Event information can be compressed by the indexer before storing it in the database. This metric collects statistics about the size of decompressed key value of a create event.
Type: Histogram
Qualification: Debug

daml.index.db.compression.exercise_argument_compressed

Summary: The size of the compressed argument of an exercise event.
Description: Event information can be compressed by the indexer before storing it in the database. This metric collects statistics about the size of compressed arguments of an exercise event.
Type: Histogram
Qualification: Debug

daml.index.db.compression.exercise_argument_uncompressed

Summary: The size of the decompressed argument of an exercise event.
Description: Event information can be compressed by the indexer before storing it in the database. This metric collects statistics about the size of decompressed arguments of an exercise event.
Type: Histogram
Qualification: Debug

daml.index.db.compression.exercise_result_compressed

Summary: The size of the compressed result of an exercise event.
Description: Event information can be compressed by the indexer before storing it in the database. This metric collects statistics about the size of compressed result of an exercise event.
Type: Histogram
Qualification: Debug

daml.index.db.compression.exercise_result_uncompressed

Summary: The size of the decompressed result of an exercise event.
Description: Event information can be compressed by the indexer before storing it in the database. This metric collects statistics about the size of compressed result of an exercise event.
Type: Histogram
Qualification: Debug

daml.index.db.flat_transactions_stream.translation

Summary: The time needed to turn serialized Daml-LF values into in-memory objects.
Description: Some index database queries that target contracts and transactions involve a Daml-LF translation step. For such queries this metric stands for the time it takes to turn the serialized Daml-LF values into in-memory representation.
Type: Timer
Qualification: Debug

daml.index.db.lookup_active_contract

Summary: The time spent fetching a contract using its id.
Description: This metric exposes the time spent fetching a contract using its id from the index db. It is then used by the Daml interpreter when evaluating a command into a transaction.
Type: Timer
Qualification: Debug

daml.index.db.lookup_key

Summary: The time spent looking up a contract using its key.
Description: This metric exposes the time spent looking up a contract using its key in the index db. It is then used by the Daml interpreter when evaluating a command into a transaction.
Type: Timer
Qualification: Debug

daml.index.db.reassignment_stream.translation

Summary: The time needed to turn serialized Daml-LF values into in-memory objects.
Description: Some index database queries that target contracts and transactions involve a Daml-LF translation step. For such queries this metric stands for the time it takes to turn the serialized Daml-LF values into in-memory representation.
Type: Timer
Qualification: Debug

daml.index.db.translation.get_lf_package

Summary: The time needed to deserialize and decode a Daml-LF archive.
Description: A Daml archive before it can be used in the interpretation needs to be deserialized and decoded, in other words converted into the in-memory representation. This metric represents time necessary to do that.
Type: Timer
Qualification: Debug

daml.index.db.tree_transactions_stream.translation

Summary: The time needed to turn serialized Daml-LF values into in-memory objects.
Description: Some index database queries that target contracts and transactions involve a Daml-LF translation step. For such queries this metric stands for the time it takes to turn the serialized Daml-LF values into in-memory representation.
Type: Timer
Qualification: Debug

daml.index.flat_transactions_buffer_size

Summary: The buffer size for flat transactions requests.
Description: An Akka stream buffer is added at the end of all streaming queries, allowing to absorb temporary downstream backpressure (e.g. when the client is slower than upstream delivery throughput). This metric gauges the size of the buffer for queries requesting flat transactions in a specific period of time that satisfy a given predicate.
Type: Counter
Qualification: Saturation

daml.index.ledger_end_sequential_id

Summary: The sequential id of the current ledger end kept in memory.
Description: The ledger end's sequential id is a monotonically increasing integer value representing the sequential id ascribed to the most recent ledger event ingested by the index db. Please note, that only a subset of all ledger events are ingested and given a sequential id. These are: creates, consuming exercises, non-consuming exercises and divulgence events. This value can be treated as a counter of all such events visible to a given participant. This metric exposes the latest ledger end's sequential id registered in the in-memory data set.
Type: Gauge

Qualification: Debug

daml.index.lf_value.compute_interface_view

Summary: The time to compute an interface view while serving transaction streams.
Description: Transaction API allows clients to request events by interface-id. When an event matches the interface - an interface view is computed, which adds to the latency. This metric represents the time for each such computation.
Type: Timer
Qualification: Debug

daml.index.package_metadata.decode_archive

Summary: The time to decode a package archive to extract metadata information.
Description: This metric represents the time spent scanning each uploaded package for new interfaces and corresponding templates.
Type: Timer
Qualification: Debug

daml.index.package_metadata.view_init

Summary: The time to initialize package metadata view.
Description: As the mapping between interfaces and templates is not persistent - it is computed for each Indexer restart by loading all packages which were ever uploaded and scanning them to extract metadata information.
Type: Timer
Qualification: Debug

daml.index.transaction_trees_buffer_size

Summary: The buffer size for transaction trees requests.
Description: An Akka stream buffer is added at the end of all streaming queries, allowing to absorb temporary downstream backpressure (e.g. when the client is slower than upstream delivery throughput). This metric gauges the size of the buffer for queries requesting transaction trees.
Type: Counter
Qualification: Saturation

daml.indexer.current_record_time_lag

Summary: The lag between the record time of a transaction and the wall-clock time registered at the ingestion phase to the index db (in milliseconds).
Description: Depending on the systemic clock skew between different machines, this value can be negative.
Type: Gauge
Qualification: Debug

daml.indexer.events*

Summary: Number of transactions processed.
Description: Represents the total number of transaction acceptance, transaction rejection, package upload, party allocation, etc. events processed.
Type: Meter
Qualification: Debug
Labels:

- **participant_id**: The id of the participant.
- **application_id**: The application generating the events.
- **event_type**: The type of ledger event processed (transaction, package upload, party allocation, configuration change).
- **status**: Indicates if the transaction was accepted or not. Possible values accepted|rejected.

daml.indexer.last_received_record_time

Summary: The time of the last event ingested by the index db (in milliseconds since EPOCH).
Description: The last received record time is a monotonically increasing integer value that represents the record time of the last event ingested by the index db. It is measured in milliseconds since the EPOCH time.
Type: Gauge
Qualification: Debug

daml.indexer.ledger_end_sequential_id

Summary: The sequential id of the current ledger end kept in the database.
Description: The ledger end's sequential id is a monotonically increasing integer value representing the sequential id ascribed to the most recent ledger event ingested by the index db. Please note, that only a subset of all ledger events are ingested and given a sequential id. These are: creates, consuming exercises, non-consuming exercises and divulgence events. This value can be treated as a counter of all such events visible to a given participant. This metric exposes the latest ledger end's sequential id registered in the database.
Type: Gauge
Qualification: Debug

daml.indexer.metered_events*

Summary: Number of ledger events that are metered.
Description: Represents the number of events that will be included in the metering report. This is an estimate of the total number and not a substitute for the metering report.
Type: Meter
Qualification: Debug
Labels:
- **participant_id**: The id of the participant.
- **application_id**: The application generating the events.

daml.lapi.streams.acs_sent

Summary: The number of the active contracts sent by the ledger api.
Description: The total number of active contracts sent over the ledger api streams to all clients.
Type: Counter
Qualification: Traffic

daml.lapi.streams.active

Summary: The number of the active streams served by the ledger api.
Description: The number of ledger api streams currently being served to all clients.
Type: Gauge
Qualification: Debug

daml.lapi.streams.completions_sent

Summary: The number of the command completions sent by the ledger api.
Description: The total number of completions sent over the ledger api streams to all clients.
Type: Counter
Qualification: Traffic

daml.lapi.streams.transaction_trees_sent

Summary: The number of the transaction trees sent over the ledger api.
Description: The total number of the transaction trees sent over the ledger api streams to all clients.
Type: Counter
Qualification: Traffic

daml.lapi.streams.transactions_sent

Summary: The number of the flat updates sent over the ledger api.
Description: The total number of the flat updates sent over the ledger api streams to all clients.
Type: Counter
Qualification: Traffic

daml.lapi.streams.update_trees_sent

Summary: The number of the update trees sent over the ledger api.
Description: The total number of the update trees sent over the ledger api streams to all clients.
Type: Counter
Qualification: Traffic

daml.parallel_indexer.input_buffer_length

Summary: The number of elements in the queue in front of the indexer.
Description: The indexer has a queue in order to absorb the back pressure and facilitate batch formation during the database ingestion.
Type: Counter
Qualification: Saturation

daml.parallel_indexer.inputmapping.batch_size

Summary: The batch sizes in the indexer.
Description: The number of state updates contained in a batch used in the indexer for database submission.
Type: Histogram
Qualification: Debug

daml.parallel_indexer.output_batched_buffer_length

Summary: The size of the queue between the indexer and the in-memory state updating flow.
Description: This counter counts batches of updates passed to the in-memory flow. Batches are dynamically-sized based on amount of backpressure exerted by the downstream stages of the flow.
Type: Counter
Qualification: Debug

daml.parallel_indexer.seqmapping.duration

Summary: The duration of the seq-mapping stage.
Description: The time that a batch of updates spends in the seq-mapping stage of the indexer.
Type: Timer
Qualification: Debug

daml.parallel_indexer.updates

Summary: The number of the state updates persisted to the database.
Description: The number of the state updates persisted to the database. There are updates such as accepted transactions, configuration changes, package uloads, party allocations, rejections, etc.
Type: Counter
Qualification: Traffic

daml.services.index.<operation>

Summary: The time to execute an index service operation.
Description: The index service is an internal component responsible for access to the index db data. Its operations are invoked whenever a client request received over the ledger api requires access to the index db. This metric captures time statistics of such operations.
Type: Timer
Qualification: Debug
Instances: get_transaction_metering, prune, configuration_entries, lookup_configuration, party_entries, list_known_parties, get_parties, get_participant_id, lookup_maximum_ledger_time, get_events_by_contract_key, get_events_by_contract_id, lookup_contract_key, lookup_contract_state_without_divulgence, lookup_active_contract, get_active_contracts, get_transaction_tree_by_id, get_transaction_by_id, transaction_trees, transactions, get_completions_limited, get_completions, latest_pruned_offsets, current_ledger_end, get_ledger_configuration, package_entries, get_lf_archive, list_lf_packages

daml.services.index.in_memory_fan_out_buffer.prune

Summary: The time to remove all elements from the in-memory fan-out buffer.
Description: It is possible to remove the oldest entries of the in-memory fan out buffer. This metric exposes the time needed to prune the buffer.
Type: Timer
Qualification: Debug

daml.services.index.in_memory_fan_out_buffer.push

Summary: The time to add a new event into the buffer.
Description: The in-memory fan-out buffer is a buffer that stores the last ingested maxBufferSize accepted and rejected submission updates as TransactionLogUpdate. It allows bypassing IndexDB persistence fetches for recent updates for flat and transaction tree streams, command completion streams and by-event-id and by-transaction-id flat and transaction tree lookups. This metric exposes the time spent on adding a new event into the buffer.
Type: Timer
Qualification: Debug

daml.services.index.in_memory_fan_out_buffer.size

Summary: The size of the in-memory fan-out buffer.
Description: The actual size of the in-memory fan-out buffer. This metric is mostly targeted for debugging purposes.
Type: Histogram
Qualification: Saturation

daml.services.read.<operation>

Summary: The time to execute a read service operation.
Description: The read service is an internal interface for reading the events from the synchronization interfaces. The metrics expose the time needed to execute each operation.
Type: Timer
Qualification: Debug
Instances: incomplete_reassignment_offsets, get_connected_domains, state_updates, get_ledger_initial_conditions

daml.services.write.<operation>

Summary: The time to execute a write service operation.
Description: The write service is an internal interface for changing the state through the synchronization services. The methods in this interface are all methods that are supported uniformly across all ledger implementations. This metric exposes the time needed to execute each operation.
Type: Timer
Qualification: Debug
Instances: prune, submit_configuration, allocate_party, upload_packages, submit_reassignment_running, submit_reassignment, submit_transaction_running, submit_transaction

daml.services.write.submit_transaction.count

Summary: The number of submitted transactions by the write service.
Description: The write service is an internal interface for changing the state through the synchronization services. The methods in this interface are all methods that are supported uniformly across all ledger implementations. This metric exposes the total number of the sumbitted transactions.
Type: Timer
Qualification: Traffic

1.29.4.2 Domain Metrics

canton.<component>.sequencer-client.application-handle

Summary: Timer monitoring time and rate of sequentially handling the event application logic
Description: All events are received sequentially. This handler records the the rate and time it takes the application (participant or domain) to handle the events.
Type: Timer
Qualification: Debug
Instances: topology-manager, mediator, sequencer

canton.<component>.sequencer-client.delay

Summary: The delay on the event processing
Description: Every message received from the sequencer carries a timestamp that was assigned by the sequencer when it sequenced the message. This timestamp is called the sequencing timestamp. The component receiving the message on the participant, mediator or topology manager side, is the sequencer client. Upon receiving the message, the sequencer client compares the time difference between the sequencing time and the computers local clock and exposes this difference as the given metric. The difference will include the clock-skew and the processing latency between assigning the timestamp on the sequencer and receiving the message by the recipient. If the difference is large compared to the usual latencies and if clock skew can be ruled out, then it means that the node is still trying to catch up with events that were sequenced by the sequencer a while ago. This can happen after having been offline for a while or if the node is too slow to keep up with the messaging load.
Type: Gauge
Qualification: Debug
Instances: topology-manager, mediator, sequencer

canton.<component>.sequencer-client.event-handle

Summary: Timer monitoring time and rate of entire event handling
Description: Most event handling cost should come from the application-handle. This timer measures the full time (which should just be marginally more than the application handle.
Type: Timer
Qualification: Debug
Instances: topology-manager, mediator, sequencer

canton.<component>.sequencer-client.load

Summary: The load on the event subscription
Description: The event subscription processor is a sequential process. The load is a factor between 0 and 1 describing how much of an existing interval has been spent in the event handler.
Type: Gauge
Qualification: Debug
Instances: topology-manager, mediator, sequencer

canton.db-storage.<service>.executor.queued

Summary: Number of database access tasks waiting in queue
Description: Database access tasks get scheduled in this queue and get executed using one of the existing asynchronous sessions. A large queue indicates that the database connection is not able to deal with the large number of requests. Note that the queue has a maximum size. Tasks that do not fit into the queue will be retried, but won't show up in this metric.
Type: Counter
Qualification: Debug
Instances: locks, write, general

canton.db-storage.<service>.executor.running

Summary: Number of database access tasks currently running
Description: Database access tasks run on an async executor. This metric shows the current number of tasks running in parallel.
Type: Counter
Qualification: Debug
Instances: locks, write, general

canton.db-storage.<service>.executor.waittime

Summary: Scheduling time metric for database tasks
Description: Every database query is scheduled using an asynchronous executor with a queue. The time a task is waiting in this queue is monitored using this metric.
Type: Timer
Qualification: Debug
Instances: locks, write, general

canton.db-storage.<storage>

Summary: Timer monitoring duration and rate of accessing the given storage
Description: Covers both read from and writes to the storage.
Type: Timer
Qualification: Debug

canton.db-storage.<storage>.load

Summary: The load on the given storage
Description: The load is a factor between 0 and 1 describing how much of an existing interval has been spent reading from or writing to the storage.
Type: Gauge
Qualification: Debug

canton.db-storage.alerts.multi-domain-event-log

Summary: Number of failed writes to the multi-domain event log
Description: Failed writes to the multi domain event log indicate an issue requiring user intervention. In the case of domain event logs, the corresponding domain no longer emits any subsequent events until domain recovery is initiated (e.g. by disconnecting and reconnecting the participant from the domain). In the case of the participant event log, an operation might need to be reissued. If this counter is larger than zero, check the canton log for errors for details.
Type: Counter
Qualification: Debug

canton.db-storage.alerts.single-dimension-event-log

Summary: Number of failed writes to the event log
Description: Failed writes to the single dimension event log indicate an issue requiring user intervention. In the case of domain event logs, the corresponding domain no longer emits any subsequent events until domain recovery is initiated (e.g. by disconnecting and reconnecting the participant from the domain). In the case of the participant event log, an operation might need to be reissued. If this counter is larger than zero, check the canton log for errors for details.
Type: Counter
Qualification: Debug

canton.mediator.max-event-age

Summary: Age of oldest unpruned mediator response.
Description: This gauge exposes the age of the oldest, unpruned mediator response in hours as a way to quantify the pruning backlog.
Type: Gauge
Qualification: Debug

canton.mediator.outstanding-requests

Summary: Number of currently outstanding requests
Description: This metric provides the number of currently open requests registered with the mediator.
Type: Gauge
Qualification: Debug

canton.mediator.requests

Summary: Number of totally processed requests
Description: This metric provides the number of totally processed requests since the system has been started.
Type: Meter
Qualification: Debug

canton.mediator.sequencer-client.handler.actual-in-flight-event-batches

Summary: Nodes process the events from the domain's sequencer in batches. This metric tracks how many such batches are processed in parallel.
Description: Incoming messages are processed by a sequencer client, which combines them into batches of size up to 'event-inbox-size' before sending them to an application handler for processing. Depending on the system's configuration, the rate at which event batches are sent to the handler may be throttled to avoid overwhelming it with too many events at once. Indicators that the configured upper bound may be too low: This metric constantly is closed to the configured maximum, which is exposed via 'max-in-flight-event-batches', while the system's resources are under-utilized. Indicators that the configured upper bound may be too high: Out-of-memory errors crashing the JVM or frequent garbage collection cycles that slow down processing. The metric tracks how many of these batches have been sent to the application handler but have not yet been fully processed. This metric can help identify potential bottlenecks or issues with the application's processing of events and provide insights into the overall workload of the system.
Type: Counter
Qualification: Saturation

canton.mediator.sequencer-client.handler.max-in-flight-event-batches

Summary: Nodes process the events from the domain's sequencer in batches. This metric tracks the upper bound of such batches being processed in parallel.
Description: Incoming messages are processed by a sequencer client, which combines them into batches of size up to 'event-inbox-size' before sending them to an application handler for processing. Depending on the system's configuration, the rate at which event batches are sent to the handler may be throttled to avoid overwhelming it with too many events at once. Configured by 'maximum-in-flight-event-batches' parameter in the sequencer-client config The metric shows the configured upper limit on how many batches the application handler may process concurrently. The metric 'actual-in-flight-event-batches' tracks the actual number of currently processed batches.
Type: Gauge

Qualification: Saturation

canton.mediator.sequencer-client.submissions.dropped

Summary: Count of send requests that did not cause an event to be sequenced
Description: Counter of send requests we did not witness a corresponding event to be sequenced by the supplied max-sequencing-time. There could be many reasons for this happening: the request may have been lost before reaching the sequencer, the sequencer may be at capacity and the the max-sequencing-time was exceeded by the time the request was processed, or the supplied max-sequencing-time may just be too small for the sequencer to be able to sequence the request.
Type: Counter
Qualification: Debug

canton.mediator.sequencer-client.submissions.in-flight

Summary: Number of sequencer send requests we have that are waiting for an outcome or timeout
Description: Incremented on every successful send to the sequencer. Decremented when the event or an error is sequenced, or when the max-sequencing-time has elapsed.
Type: Counter
Qualification: Debug

canton.mediator.sequencer-client.submissions.overloaded

Summary: Count of send requests which receive an overloaded response
Description: Counter that is incremented if a send request receives an overloaded response from the sequencer.
Type: Counter
Qualification: Debug

canton.mediator.sequencer-client.submissions.sends

Summary: Rate and timings of send requests to the sequencer
Description: Provides a rate and time of how long it takes for send requests to be accepted by the sequencer. Note that this is just for the request to be made and not for the requested event to actually be sequenced.
Type: Timer
Qualification: Debug

canton.mediator.sequencer-client.submissions.sequencing

Summary: Rate and timings of sequencing requests
Description: This timer is started when a submission is made to the sequencer and then completed when a corresponding event is witnessed from the sequencer, so will encompass the entire duration for the sequencer to sequence the request. If the request does not result in an event no timing will be recorded.
Type: Timer
Qualification: Debug

canton.sequencer.db-storage.<storage>

Summary: Timer monitoring duration and rate of accessing the given storage
Description: Covers both read from and writes to the storage.
Type: Timer
Qualification: Debug

canton.sequencer.db-storage.<storage>.load

Summary: The load on the given storage
Description: The load is a factor between 0 and 1 describing how much of an existing interval has been spent reading from or writing to the storage.
Type: Gauge
Qualification: Debug

canton.sequencer.db-storage.alerts.multi-domain-event-log

Summary: Number of failed writes to the multi-domain event log
Description: Failed writes to the multi domain event log indicate an issue requiring user intervention. In the case of domain event logs, the corresponding domain no longer emits any subsequent events until domain recovery is initiated (e.g. by disconnecting and reconnecting the participant from the domain). In the case of the participant event log, an operation might need to be reissued. If this counter is larger than zero, check the canton log for errors for details.
Type: Counter
Qualification: Debug

canton.sequencer.db-storage.alerts.single-dimension-event-log

Summary: Number of failed writes to the event log
Description: Failed writes to the single dimension event log indicate an issue requiring user intervention. In the case of domain event logs, the corresponding domain no longer emits any subsequent events until domain recovery is initiated (e.g. by disconnecting and reconnecting the participant from the domain). In the case of the participant event log, an operation might need to be reissued. If this counter is larger than zero, check the canton log for errors for details.
Type: Counter
Qualification: Debug

canton.sequencer.db-storage.general.executor.queued

Summary: Number of database access tasks waiting in queue
Description: Database access tasks get scheduled in this queue and get executed using one of the existing asynchronous sessions. A large queue indicates that the database connection is not able to deal with the large number of requests. Note that the queue has a maximum size. Tasks that do not fit into the queue will be retried, but won't show up in this metric.
Type: Counter
Qualification: Debug

canton.sequencer.db-storage.general.executor.running

Summary: Number of database access tasks currently running
Description: Database access tasks run on an async executor. This metric shows the current number of tasks running in parallel.
Type: Counter
Qualification: Debug

canton.sequencer.db-storage.general.executor.waittime

Summary: Scheduling time metric for database tasks
Description: Every database query is scheduled using an asynchronous executor with a queue. The time a task is waiting in this queue is monitored using this metric.
Type: Timer
Qualification: Debug

canton.sequencer.db-storage.locks.executor.queued

Summary: Number of database access tasks waiting in queue
Description: Database access tasks get scheduled in this queue and get executed using one of the existing asynchronous sessions. A large queue indicates that the database connection is not able to deal with the large number of requests. Note that the queue has a maximum size. Tasks that do not fit into the queue will be retried, but won't show up in this metric.
Type: Counter
Qualification: Debug

canton.sequencer.db-storage.locks.executor.running

Summary: Number of database access tasks currently running
Description: Database access tasks run on an async executor. This metric shows the current number of tasks running in parallel.
Type: Counter
Qualification: Debug

canton.sequencer.db-storage.locks.executor.waittime

Summary: Scheduling time metric for database tasks
Description: Every database query is scheduled using an asynchronous executor with a queue. The time a task is waiting in this queue is monitored using this metric.
Type: Timer
Qualification: Debug

canton.sequencer.db-storage.write.executor.queued

Summary: Number of database access tasks waiting in queue
Description: Database access tasks get scheduled in this queue and get executed using one of the existing asynchronous sessions. A large queue indicates that the database connection is not able to deal with the large number of requests. Note that the queue has a maximum size. Tasks that do not fit into the queue will be retried, but won't show up in this metric.
Type: Counter
Qualification: Debug

canton.sequencer.db-storage.write.executor.running

Summary: Number of database access tasks currently running
Description: Database access tasks run on an async executor. This metric shows the current number of tasks running in parallel.
Type: Counter
Qualification: Debug

canton.sequencer.db-storage.write.executor.waittime

Summary: Scheduling time metric for database tasks
Description: Every database query is scheduled using an asynchronous executor with a queue. The time a task is waiting in this queue is monitored using this metric.
Type: Timer
Qualification: Debug

canton.sequencer.max-event-age

Summary: Age of oldest unpruned sequencer event.
Description: This gauge exposes the age of the oldest, unpruned sequencer event in hours as a way to quantify the pruning backlog.
Type: Gauge
Qualification: Debug

canton.sequencer.processed

Summary: Number of messages processed by the sequencer
Description: This metric measures the number of successfully validated messages processed by the sequencer since the start of this process.
Type: Meter
Qualification: Debug

canton.sequencer.processed-bytes

Summary: Number of message bytes processed by the sequencer
Description: This metric measures the total number of message bytes processed by the sequencer. If the message received by the sequencer contains duplicate or irrelevant fields, the contents of these fields do not contribute to this metric.
Type: Meter
Qualification: Debug

canton.sequencer.sequencer-client.handler.actual-in-flight-event-batches

Summary: Nodes process the events from the domain's sequencer in batches. This metric tracks how many such batches are processed in parallel.
Description: Incoming messages are processed by a sequencer client, which combines them into batches of size up to 'event-inbox-size' before sending them to an application handler for processing. Depending on the system's configuration, the rate at which event batches are sent to the handler may be throttled to avoid overwhelming it with too many events at once. Indicators that the configured upper bound may be too low: This metric constantly is closed to the configured maximum, which is exposed via 'max-in-flight-event-batches', while the system's resources are under-utilized. Indicators that the configured upper bound may be too high: Out-of-memory errors crashing the JVM or frequent garbage collection cycles that slow down processing. The metric tracks how many of these batches have been sent to the application handler but have not yet been fully processed. This metric can help identify potential bottlenecks or issues with the application's processing of events and provide insights into the overall workload of the system.
Type: Counter
Qualification: Saturation

canton.sequencer.sequencer-client.handler.max-in-flight-event-batches

Summary: Nodes process the events from the domain's sequencer in batches. This metric tracks the upper bound of such batches being processed in parallel.
Description: Incoming messages are processed by a sequencer client, which combines them into batches of size up to 'event-inbox-size' before sending them to an application handler for processing. Depending on the system's configuration, the rate at which event batches are sent to the handler may be throttled to avoid overwhelming it with too many events at once. Configured by 'maximum-in-flight-event-batches' parameter in the sequencer-client config The metric shows the configured upper limit on how many batches the application handler may process concurrently. The metric 'actual-in-flight-event-batches' tracks the actual number of currently processed batches.

Type: Gauge
Qualification: Saturation

canton.sequencer.sequencer-client.submissions.dropped

Summary: Count of send requests that did not cause an event to be sequenced
Description: Counter of send requests we did not witness a corresponding event to be sequenced by the supplied max-sequencing-time. There could be many reasons for this happening: the request may have been lost before reaching the sequencer, the sequencer may be at capacity and the the max-sequencing-time was exceeded by the time the request was processed, or the supplied max-sequencing-time may just be too small for the sequencer to be able to sequence the request.
Type: Counter
Qualification: Debug

canton.sequencer.sequencer-client.submissions.in-flight

Summary: Number of sequencer send requests we have that are waiting for an outcome or timeout
Description: Incremented on every successful send to the sequencer. Decremented when the event or an error is sequenced, or when the max-sequencing-time has elapsed.
Type: Counter
Qualification: Debug

canton.sequencer.sequencer-client.submissions.overloaded

Summary: Count of send requests which receive an overloaded response
Description: Counter that is incremented if a send request receives an overloaded response from the sequencer.
Type: Counter
Qualification: Debug

canton.sequencer.sequencer-client.submissions.sends

Summary: Rate and timings of send requests to the sequencer
Description: Provides a rate and time of how long it takes for send requests to be accepted by the sequencer. Note that this is just for the request to be made and not for the requested event to actually be sequenced.
Type: Timer
Qualification: Debug

canton.sequencer.sequencer-client.submissions.sequencing

Summary: Rate and timings of sequencing requests
Description: This timer is started when a submission is made to the sequencer and then completed when a corresponding event is witnessed from the sequencer, so will encompass the entire duration for the sequencer to sequence the request. If the request does not result in an event no timing will be recorded.
Type: Timer
Qualification: Debug

canton.sequencer.subscriptions

Summary: Number of active sequencer subscriptions
Description: This metric indicates the number of active subscriptions currently open and actively served subscriptions at the sequencer.
Type: Gauge
Qualification: Debug

canton.sequencer.time-requests

Summary: Number of time requests received by the sequencer
Description: When a Participant needs to know the domain time it will make a request for a time proof to be sequenced. It would be normal to see a small number of these being sequenced, however if this number becomes a significant portion of the total requests to the sequencer it could indicate that the strategy for requesting times may need to be revised to deal with different clock skews and latencies between the sequencer and participants.
Type: Meter
Qualification: Debug

canton.topology-manager.sequencer-client.handler.actual-in-flight-event-batches

Summary: Nodes process the events from the domain's sequencer in batches. This metric tracks how many such batches are processed in parallel.
Description: Incoming messages are processed by a sequencer client, which combines them into batches of size up to 'event-inbox-size' before sending them to an application handler for processing. Depending on the system's configuration, the rate at which event batches are sent to the handler may be throttled to avoid overwhelming it with too many events at once. Indicators that the configured upper bound may be too low: This metric constantly is closed to the configured maximum, which is exposed via 'max-in-flight-event-batches', while the system's resources are under-utilized. Indicators that the configured upper bound may be too high: Out-of-memory errors crashing the JVM or frequent garbage collection cycles that slow down processing. The metric tracks how many of these batches have been sent to the application handler but have not yet been fully processed. This metric can help identify potential bottlenecks or issues with the application's processing of events and provide insights into the overall workload of the system.
Type: Counter
Qualification: Saturation

canton.topology-manager.sequencer-client.handler.max-in-flight-event-batches

Summary: Nodes process the events from the domain's sequencer in batches. This metric tracks the upper bound of such batches being processed in parallel.
Description: Incoming messages are processed by a sequencer client, which combines them into batches of size up to 'event-inbox-size' before sending them to an application handler for processing. Depending on the system's configuration, the rate at which event batches are sent to the handler may be throttled to avoid overwhelming it with too many events at once. Configured by 'maximum-in-flight-event-batches' parameter in the sequencer-client config The metric shows the configured upper limit on how many batches the application handler may process concurrently. The metric 'actual-in-flight-event-batches' tracks the actual number of currently processed batches.
Type: Gauge
Qualification: Saturation

canton.topology-manager.sequencer-client.submissions.dropped

Summary: Count of send requests that did not cause an event to be sequenced
Description: Counter of send requests we did not witness a corresponding event to be sequenced by the supplied max-sequencing-time. There could be many reasons for this happening: the request may have been lost before reaching the sequencer, the sequencer may be at capacity and the the max-sequencing-time was exceeded by the time the request was processed, or the supplied max-sequencing-time may just be too small for the sequencer to be able to sequence the request.
Type: Counter
Qualification: Debug

canton.topology-manager.sequencer-client.submissions.in-flight

Summary: Number of sequencer send requests we have that are waiting for an outcome or timeout
Description: Incremented on every successful send to the sequencer. Decremented when the event or an error is sequenced, or when the max-sequencing-time has elapsed.
Type: Counter
Qualification: Debug

canton.topology-manager.sequencer-client.submissions.overloaded

Summary: Count of send requests which receive an overloaded response
Description: Counter that is incremented if a send request receives an overloaded response from the sequencer.
Type: Counter
Qualification: Debug

canton.topology-manager.sequencer-client.submissions.sends

Summary: Rate and timings of send requests to the sequencer
Description: Provides a rate and time of how long it takes for send requests to be accepted by the sequencer. Note that this is just for the request to be made and not for the requested event to actually be sequenced.
Type: Timer
Qualification: Debug

canton.topology-manager.sequencer-client.submissions.sequencing

Summary: Rate and timings of sequencing requests
Description: This timer is started when a submission is made to the sequencer and then completed when a corresponding event is witnessed from the sequencer, so will encompass the entire duration for the sequencer to sequence the request. If the request does not result in an event no timing will be recorded.
Type: Timer
Qualification: Debug

1.29.4.3 Health Metrics

The following metrics are exposed for all components.

daml_health_status

Description: The status of the component
Values:
- **0**: Not healthy
- **1**: Healthy

Labels:
- **component**: the name of the component being monitored

Type: Gauge

1.29.4.4 gRPC Metrics

The following metrics are exposed for all gRPC endpoints. These metrics have the following common labels attached:

grpc_service_name: fully qualified name of the gRPC service (e.g. `com.daml.ledger.api.v1.ActiveContractsService`)
grpc_method_name: name of the gRPC method (e.g. `GetActiveContracts`)
grpc_client_type: type of client connection (`unary` or `streaming`)
grpc_server_type: type of server connection (`unary` or `streaming`)
service: Canton service's name (e.g. `participant`, `sequencer`, etc.)

daml_grpc_server_duration_seconds

Description: Distribution of the durations of serving gRPC requests
Type: Histogram

daml_grpc_server_messages_sent_total

Description: Total number of gRPC messages sent (on either type of connection)
Type: Counter

daml_grpc_server_messages_received_total

Description: Total number of gRPC messages received (on either type of connection)
Type: Counter

daml_grpc_server_started_total

Description: Total number of started gRPC requests (on either type of connection)
Type: Counter

daml_grpc_server_handled_total

Description: Total number of handled gRPC requests
Labels:
- **grpc_code**: returned gRPC status code for the call (`OK`, `CANCELLED`, `INVALID_ARGUMENT`, etc.)

Type: Counter

daml_grpc_server_messages_sent_bytes

Description: Distribution of payload sizes in gRPC messages sent (both unary and streaming)
Type: Histogram

daml_grpc_server_messages_received_bytes

Description: Distribution of payload sizes in gRPC messages received (both unary and streaming)
Type: Histogram

1.29.4.5 HTTP Metrics

The following metrics are exposed for all HTTP endpoints. These metrics have the following common labels attached:

http_verb: HTTP verb used for a given call (e.g. `GET` or `PUT`)
host: fully qualified hostname of the HTTP endpoint (e.g. `example.com`)
path: path of the HTTP endpoint (e.g. `/v1/parties/create`)
service: Daml service's name (`json_api` for the HTTP JSON API Service)

daml_http_requests_duration_seconds

Description: Distribution of the durations of serving HTTP requests
Type: Histogram

daml_http_requests_total

Description: Total number of HTTP requests completed
Labels:
- **http_status**: returned HTTP status code for the call

Type: Counter

daml_http_websocket_messages_received_total

Description: Total number of WebSocket messages received
Type: Counter

daml_http_websocket_messages_sent_total

Description: Total number of WebSocket messages sent
Type: Counter

daml_http_requests_payload_bytes

Description: Distribution of payload sizes in HTTP requests received
Type: Histogram

daml_http_responses_payload_bytes

Description: Distribution of payload sizes in HTTP responses sent
Type: Histogram

daml_http_websocket_messages_received_bytes

Description: Distribution of payload sizes in WebSocket messages received
Type: Histogram

daml_http_websocket_messages_sent_bytes

Description: Distribution of payload sizes in WebSocket messages sent
Type: Histogram

1.29.4.6 Java Execution Service Metrics

The following metrics are exposed for all execution services used by Daml components. These metrics have the following common labels attached:

name: The name of the executor service, that identifies its internal usage
type: The type of the execution service: fork_join and thread_pool are supported

daml_executor_pool_size

Description: Number of worker threads present in the pool
Type: Gauge

daml_executor_pool_core

Description: Core number of threads
Type: Gauge
Observation: Only available for *type = thread_pool*

daml_executor_pool_max

Description: Maximum allowed number of threads
Type: Gauge
Observation: Only available for *type = thread_pool*

daml_executor_pool_largest

Description: Largest number of threads that have ever simultaneously been in the pool
Type: Gauge
Observation: Only available for *type* = *thread_pool*

daml_executor_threads_active

Description: Estimate of the number of threads that are executing tasks
Type: Gauge

daml_executor_threads_running

Description: Estimate of the number of worker threads that are not blocked waiting to join tasks or for other managed synchronization
Type: Gauge
Observation: Only available for *type* = *fork_join*

daml_executor_tasks_queued

Description: Approximate number of tasks that are queued for execution
Type: Gauge

daml_executor_tasks_executing_queued

Description: Estimate of the total number of tasks currently held in queues by worker threads (but not including tasks submitted to the pool that have not begun executing)
Type: Gauge
Observation: Only available for *type* = *fork_join*

daml_executor_tasks_stolen

Description: Estimate of the total number of completed tasks that were executed by a thread other than their submitter
Type: Gauge
Observation: Only available for *type* = *fork_join*

daml_executor_tasks_submitted

Description: Approximate total number of tasks that have ever been scheduled for execution
Type: Gauge
Observation: Only available for *type* = *thread_pool*

daml_executor_tasks_completed

Description: Approximate total number of tasks that have completed execution
Type: Gauge
Observation: Only available for *type* = *thread_pool*

daml_executor_tasks_queue_remaining

Description: Additional elements that this queue can ideally accept without blocking
Type: Gauge
Observation: Only available for *type* = *thread_pool*

1.29.4.7 Pruning Metrics

The following metrics are exposed for all pruning processes. These metrics have the following labels:

phase: The name of the pruning phase being monitored

daml_services_pruning_prune_started_total

Description: Total number of started pruning processes
Type: Counter

daml_services_pruning_prune_completed_total

Description: Total number of completed pruning processes
Type: Counter

1.29.4.8 JVM Metrics

The following metrics are exposed for the JVM, if enabled.

runtime_jvm_gc_time

Description: Time spent in a given JVM garbage collector in milliseconds
Labels:
- **gc**: Garbage collector regions (eg: `G1 Old Generation`, `G1 New Generation`)

Type: Counter

runtime_jvm_gc_count

Description: The number of collections that have occurred for a given JVM garbage collector
Labels:
- **gc**: Garbage collector regions (eg: `G1 Old Generation`, `G1 New Generation`)

Type: Counter

runtime_jvm_memory_area

Description: JVM memory area statistics
Labels:
- **area**: Can be `heap` or `non_heap`
- **type**: Can be `committed`, `used` or `max`

runtime_jvm_memory_pool

Description: JVM memory pool statistics
Labels:
- **pool**: Defined pool name.
- **type**: Can be `committed`, `used` or `max`

1.29.5 Logging

Canton uses Logback as the logging library. All Canton logs derive from the logger `com.digitalasset.canton`. By default, Canton will write a log to the file `log/canton.log` using the `INFO` log-level and will also log `WARN` and `ERROR` to stdout.

How Canton produces log files can be configured extensively on the command line using the following options:

`-v` (or `--verbose`) is a short option to set the Canton log level to `DEBUG`. This is likely the most common log option you will use.
`--debug` sets all log levels except stdout to `DEBUG`. Stdout is set to `INFO`. Note that `DEBUG` logs of external libraries can be very noisy.
`--log-level-root=<level>` configures the log-level of the root logger. This changes the log level of Canton and of external libraries, but not of stdout.
`--log-level-canton=<level>` configures the log-level of only the Canton logger.
`--log-level-stdout=<level>` configures the log-level of stdout. This will usually be the text displayed in the Canton console.
`--log-file-name=log/canton.log` configures the location of the log file.

`--log-file-appender=flat|rolling|off` configures if and how logging to a file should be done. The rolling appender will roll the files according to the defined date-time pattern.
`--log-file-rolling-history=12` configures the number of historical files to keep when using the rolling appender.
`--log-file-rolling-pattern=YYYY-mm-dd` configures the rolling file suffix (and therefore the frequency) of how files should be rolled.
`--log-truncate` configures whether the log file should be truncated on startup.
`--log-profile=container` provides a default set of logging settings for a particular setup. Only the `container` profile is supported, which logs to STDOUT. It turns off flat file logging to avoid storage leaks due to log files within a container.
`--log-immediate-flush=false` turns off immediate flushing of the log output to the log file.

Note that if you use `--log-profile`, the order of the command line arguments matters. The profile settings can be overridden on the command line by placing adjustments after the profile has been selected.

Canton supports the normal log4j logging levels: `TRACE`, `DEBUG`, `INFO`, `WARN`, and `ERROR`.

For further customization, a custom logback configuration can be provided using `JAVA_OPTS`.

```
JAVA_OPTS="-Dlogback.configurationFile=./path-to-file.xml" ./bin/canton --config .
↪..
```

If you use a custom log-file, the command line arguments for logging will not have any effect, except that `--log-level-canton` and `--log-level-root` can still be used to adjust the log level of the root loggers.

1.29.5.1 Viewing Logs

A log file viewer such as lnav is recommended to view Canton logs and resolve issues. Among other features, lnav has automatic syntax highlighting, convenient filtering for specific log messages, and the ability to view log files of different Canton components in a single view. This makes viewing logs and resolving issues more efficient than using standard UNIX tools such as less or grep.

The following features are especially useful when using `lnav`:

Viewing log files of different Canton components in a single view, merged according to timestamps (`lnav <log1> <log2> ...`).
Filtering specific log messages in (`:filter-in <regex>`) or out (`:filter-out <regex>`). When filtering messages (for example, with a given trace-id), a transaction can be traced across different components, especially when using the single-view-feature described earlier.
Searching for specific log messages (`/<regex>`) and jumping between them (`n` and `N`).
Automatic syntax highlighting of parts of log messages (such as timestamps) and log messages themselves (for example, `WARN` log messages are yellow).
Jumping between error (`e` and `E`) and warn messages (`w` and `W`).
Selectively activating and deactivating different filters and files (`TAB` and ``` to activate/deactivate a filter).
Marking lines (`m`) and jumping back and forth between marked lines (`u` and `U`).
Jumping back and forth between lines that have the same *trace-id* (`o` and `O`).

The custom lnav log format file for Canton logs `canton.lnav.json` is bundled in any Canton release. You can install it with `lnav -i canton.lnav.json`. JSON-based log files (which need to use the file suffix `.clog`) can be viewed using the `canton-json.lnav.json` format file.

1.29.5.2 Detailed Logging

By default, logging omits details to avoid writing sensitive data into log files. For debugging or educational purposes, you can turn on additional logging using the following configuration switches:

```
canton.monitoring.logging {
    event-details = true
    api {
        message-payloads = true
        max-method-length = 1000
        max-message-lines = 10000
        max-string-length = 10000
        max-metadata-size = 10000
    }
}
```

This turns on payload logging in the `ApiRequestLogger`, which records every gRPC API invocation, and turns on detailed logging of the `SequencerClient` and the transaction trees. Please note that all additional events are logged at `DEBUG` level.

Note: Note that the detailed event logging will happen within an gRPC API Interceptor. This creates a sequential bottleneck as every message that is sent or received gets translated into a pretty-printed string. You will not be able to achieve the same performance if this setting is turned on.

1.29.6 Tracing

For further debugging, Canton provides a trace-id which allows you to trace the processing of requests through the system. The trace-id is exposed to logback through the *mapping diagnostic context* and can be included in the logback output pattern using `%mdc{trace-id}`.

The trace-id propagation is enabled by setting the `canton.monitoring.tracing.propagation = enabled` configuration option, which is enabled by default.

You can configure the service where traces and spans are reported for observing distributed traces. Refer to *Traces* for a preview.

Jaeger and Zipkin are supported. For example, Jaeger reporting can be configured as follows:

```
monitoring.tracing.tracer.exporter {
  type = jaeger
  address = ... // default: "localhost"
  port = ... // default: 14250
}
```

This configuration connects to a running Jaeger server to report tracing information.

You can run Jaeger in a Docker container as follows:

```
docker run --rm -it --name jaeger\
  -p 16686:16686 \
  -p 14250:14250 \
  jaegertracing/all-in-one:1.22.0
```

If you prefer not to use Docker, you can download the binary for your specific OS at Download Jaeger. Unzip the file and then run the binary *jaeger-all-in-one* (no arguments are needed). By default, Jaeger will expose port *16686* (for its UI, which can be seen in a browser window) and port *14250* (to which Canton will report trace information). Be sure to properly expose these ports.

Make sure that all Canton nodes in the network report to the same Jaeger server to have an accurate view of the full traces. Also, ensure that the Jaeger server is reachable by all Canton nodes.

1.29.6.1 Sampling

You can change how often spans are sampled and reported to the configured exporter. By default, it will always report (`monitoring.tracing.tracer.sampler.type = always-on`). You can configure it to never report (`monitoring.tracing.tracer.sampler.type = always-off`), although this is less useful. Also, you can configure only a specific fraction of spans to be reported as follows:

```
monitoring.tracing.tracer.sampler = {
  type = trace-id-ratio
  ratio = 0.5
}
```

You can also change the parent-based sampling property. By default, it is turned on (`monitoring.tracing.tracer.sampler.parent-based = true`). When turned on, a span is sampled iff its parent is sampled (the root span will follow the configured sampling strategy). There will never be incomplete traces; either the full trace is sampled or it is not. If you change this property, all spans will follow the configured sampling strategy and ignore whether the parent is sampled.

1.29.6.2 Known Limitations

Not every trace created which can be observed in logs is reported to the configured trace collector service. Traces originated at console commands or that are part of the transaction protocol are largely reported, while other types of traces are added to the set of reported traces as the need arises.

Also, the transaction protocol trace has a known limitation: once a command is submitted and its trace is fully reported, a new trace is created for any resulting Daml events that are processed. This occurs because the ledger API does not propagate any trace context information from the command submission to the transaction subscription. As an example, when a participant creates a `Ping` contract, you can see the full transaction processing trace of the `Ping` command being submitted. However, a participant that processes the `Ping` by exercising `Respond` and creating a `Pong` contract creates a separate trace instead of using the same one.

This differs from a situation where a single Daml transaction results in multiple actions at the same time, such as archiving and creating multiple contracts. In that case, a single trace encompasses the entire process, since it occurs as part of a single transaction rather than the result of an external process reacting to Daml events.

1.29.6.3 Traces

Traces contain operations that are each represented by a span. A trace is a directed acyclic graph (DAG) of spans, where the edges between spans are defined as parent/child relationships (the definitions come from the Opentelemetry glossary).

Canton reports several types of traces. One example: every Canton console command that interacts with the Admin API starts a trace whose initial span last for the entire duration of the command, including the gRPC call to the specific Admin API endpoint.

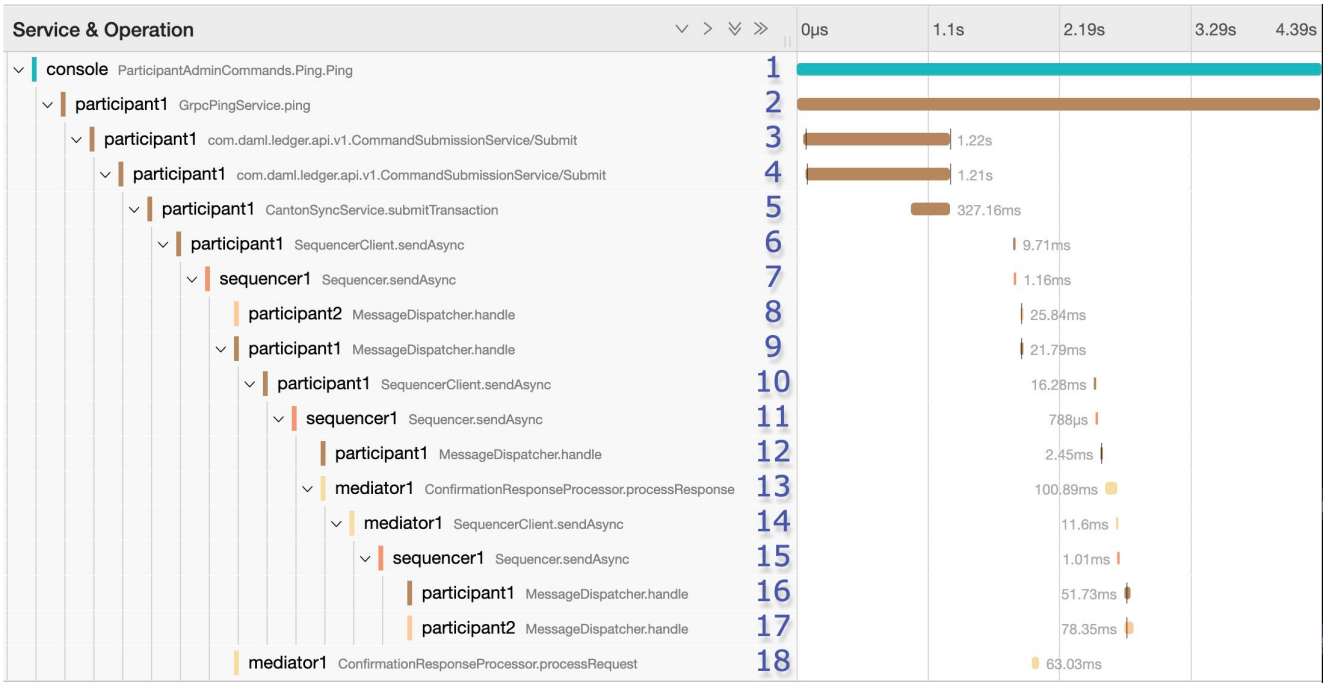


Fig. 18: Graph of a Canton ping trace containing 18 spans

Traces of Daml command submissions are important. The trace illustrated in the figure results when you perform a Canton ping using the console. The ping is a smoke test that sends a Daml transaction (create Ping, exercise choice Pong, exercise choice Archive) to test a connection. It uses a particular smart contract that is preinstalled on every Canton participant. The command uses the Admin API to access a preinstalled application, which then issues Ledger API commands operating on this smart contract. In this example, the trace contains 18 spans. The ping is started by `participant1`, and `participant2` is the target. The trace focuses on the message exchange through the sequencer without digging deep into the message handlers or further processing of transactions.

In some cases, spans may start later than the end of their parents, due to asynchronous processing. This typically occurs when a new operation is placed on a queue to be handled later, which immediately frees the parent span and ends it.

The initial span (span 1) covers the duration of the ping operation. In span 2, the GrpcPingService in the participant node handles a gRPC request made by the console. It also lasts for the duration of the ping operation.

The Canton ping consists of three Daml commands:

1. The admin party for `participant1` creates a `Ping` contract.

2. The admin party for `participant2` exercises the `Respond` consuming choice on the contract, which results in the creation of a `Pong` contract.
3. The admin party for `participant1` exercises the `Ack` consuming choice on it.

The submission of the first of the three Daml commands (the creation of the Ping contract) starts at span 3 in the example trace. Due to a limitation explained in the next section, the other two Daml command submissions are not linked to this trace. It is possible to find them separately. In any case, span 2 will only complete once the three Daml commands are completed.

At span 3, the participant node is on the client side of the ledger API. In other use cases, it could be an application integrated with the participant. This span lasts for the duration of the gRPC call, which is received on the server side in span 4 and handled by the `CantonSyncService` in span 5. The request is then received and acknowledged, but not fully processed. It is processed asynchronously later, which means that spans 3 through 5 will complete before the request is handled.

Missing steps from the trace (which account for part of the gap between spans 5 and 6) are:

> The domain routing where the participant decides which domain to use for the command submission.
> The preparation of the initial set of messages to be sent.

The start of the Canton transaction protocol begins at span 6. In this span, `participant1` sends a request to `sequencer1` to sequence the initial set of confirmation request messages as part of phase 1 of the transaction protocol. The transaction protocol has *seven phases*.

At span 7, `sequencer1` receives the request and registers it. Receipt of the messages is not part of this span. That happens asynchronously at a later point.

At span 18, as part of phase 2, `mediator1` receives an informee message. It only needs to validate and register it. Since it doesn't need to respond, span 18 has no children.

As part of phase 3, `participant2` receives a message (see span 8), and `participant1` also receives a message (see span 9). Both participants asynchronously validate the messages. `participant2` does not need to respond. Since it is only an observer, span 8 has no children. `participant1` responds, however, which is visible at span 10. There, it again makes a call to `sequencer1`, which receives it at span 11.

At span 12, `participant1` receives a successful send response message that signals that its message to the mediator was successfully sequenced. This occurs as part of phase 4, where confirmation responses are sent to the mediator. The mediator receives it at span 13, and it validates the message (phase 5).

In spans 14 and 15, `mediator1` (now at phase 6) asks `sequencer1` to send the transaction result messages to the participants.

To end this round of the transaction protocol, `participant1` and `participant2` receive their messages at spans 16 and 17, respectively. The messages are asynchronously validated, and their projections of the virtual shared ledger are updated (phase 7).

As mentioned, there are two other transaction submissions that are unlinked from this ping trace but are part of the operation. The second one starts at a span titled `admin-ping.processTransaction`, which is created by `participant2`. The third one has the same name but is initiated by `participant1`.

1.29.7 Node Health Status

Each Canton node exposes rich health status information. Running:

```
<node>.health.status
```

returns a status object, which can be one of:

`Failure`: if the status of the node cannot be determined, including an error message of why it failed
`NotInitialized`: if the node is not yet initialized
`Success[NodeStatus]`: if the status could be determined, including the detailed status

The `NodeStatus` differs depending on the node type. A participant node responds with a message containing:

`Participant id`: the participant id of the node
`Uptime`: the uptime of this node
`Ports`: the ports on which the participant node exposes the Ledger and the Admin API.
`Connected domains`: the list of domains to which the participant is properly connected
`Unhealthy domains`: the list of domains to which the participant is trying to connect, but the connection is not ready for command submission
`Active`: true if this instance is the active replica (It can be false in the case of the passive instance of a high-availability deployment.)

A domain node or a sequencer node responds with a message containing:

`Domain id`: the unique identifier of the domain
`Uptime`: the uptime of this node
`Ports`: the ports on which the domain node exposes the Public and the Admin API
`Connected Participants`: the list of connected participants
`Sequencer`: a boolean flag indicating whether the embedded sequencer writer is operational

A domain topology manager or a mediator node returns:

`Node uid`: the unique identifier of the node
`Uptime`: the uptime of this node
`Ports`: the ports on which the node hosts its APIs
`Active`: true if this instance is the active replica (It can be false in the case of the passive instance of a high-availability deployment.)

Additionally, all nodes also return a `components` field detailing the health state of each of its internal runtime dependencies. The actual components differ per node and can give further insights into the node's current status. Example components include storage access, domain connectivity, and sequencer backend connectivity.

1.29.8 Health Checks

1.29.8.1 gRPC Health Check Service

Each Canton node can optionally be configured to start a gRPC server exposing the gRPC Health Service. Passive nodes (see *High Availability* for more information on active/passive states) return `NOT_SERVING`. Consider this when configuring liveness and readiness probes in a Kubernetes environment.

The precise way the state is computed is subject to change.

Here is an example monitoring configuration to place inside a node configuration object:

```
monitoring.grpc-health-server {
  address = "127.0.0.1"
  port = 5861
}
```

Note: The gRPC health server is configured per Canton node, not per process, as is the case for the HTTP health check server (see below). This means that the configuration must be inserted within a node's configuration object.

Note: To support usage as a Kubernetes liveness probe, the health server exposes a service named `liveness` that should be targeted when configuring a gRPC probe. The latter service always returns `SERVING`.

1.29.8.2 HTTP Health Check

Optionally, the `canton` process can expose an HTTP endpoint indicating whether the process believes it is healthy. This may be used as an uptime check or as a Kubernetes liveness probe. If enabled, the `/health` endpoint will respond to a `GET` HTTP request with a 200 HTTP status code (if healthy) or 500 (if unhealthy, along with a plain text description of why it is unhealthy).

To enable this health endpoint, add a `monitoring` section to the Canton configuration. Since this health check is for the whole process, add it directly to the `canton` configuration rather than for a specific node.

```
canton {
  monitoring.health {
   server {
      port = 7000
   }

   check {
     type = ping
     participant = participant1
     interval = 30s
   }
}
```

This health check causes `participant1` to ledger ping itself every 30 seconds. The process is considered healthy if the ping is successful.

1.29.9 Health Dumps

You should provide as much information as possible to receive efficient support. For this purpose, Canton implements an information-gathering facility that gathers key essential system information for support staff. If you encounter an error where you need assistance, please ensure the following:

Start Canton in interactive mode, with the `-v` option to enable debug logging: `./bin/canton -v -c <myconfig>`. This provides a console prompt.
Reproduce the error by following the steps that previously caused the error. Write down these steps so they can be provided to support staff.
After you observe the error, type `health.dump()` into the Canton console to generate a ZIP file.

This creates a dump file (`.zip`) that stores the following information:

The configuration you are using, with all sensitive data stripped from it (no passwords).
An extract of the log file. Sensitive data is not logged into log files.
A current snapshot on Canton metrics.
A stacktrace for each running thread.

Provide the gathered information to your support contact together with the exact list of steps that led to the issue. Providing complete information is very important to help troubleshoot issues.

1.29.9.1 Remote Health Dumps

When running a console configured to access remote nodes, the `health.dump()` command gathers health data from the remote nodes and packages them into resulting zip files. There is no special action required. You can obtain the health data of a specific node by targeting it when running the command. For example:

```
remoteParticipant1.health.dump()
```

When packaging large amounts of data, increase the default timeout of the dump command:

```
health.dump(timeout = 2.minutes)
```

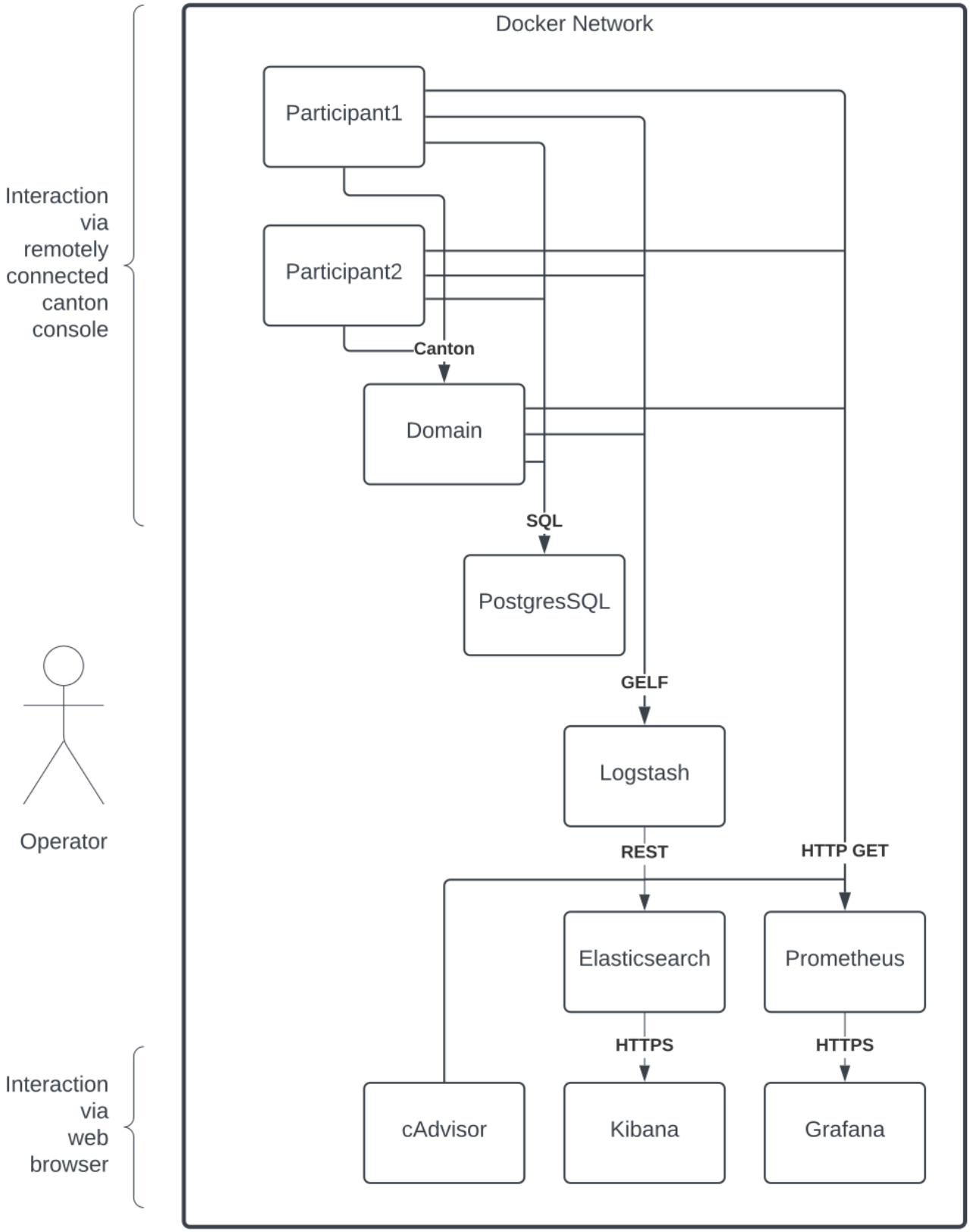
1.29.10 Example Monitoring Setup

This section provides an example of how Canton can be run inside a connected network of Docker containers. The example also shows how you can monitor network activity. See the monitoring glossary for an explanation of the terms and the *Monitoring Choices* section for the reasoning behind the example monitoring setup.

1.29.10.1 Container Setup

To configure Docker Compose to spin up the Docker container network shown in the diagram, use the information below. See the *compose* documentation for detailed information concerning the structure of the configuration files.

compose allows you to provide the overall configuration across multiple files. Each configuration file is described below, followed by information on how to bring them together in a running network.


Docker Network
Participant1
Participant2
Canton
Domain
SQL
PostgresSQL
GELF
Logstash
REST
HTTP GET
Elasticsearch
Prometheus
HTTPS
HTTPS
cAdvisor
Kibana
Grafana
Interaction via remotely connected canton console
Operator
Interaction via web browser

Intended Use

This example is intended to demonstrate how to expose, aggregate, and observe monitoring information from Canton. It is not suitable for production without alterations. Note the following warnings:

> **Warning:** Ports are exposed from the Docker network that are not necessary to support the UI. For example, the network can allow low-level interaction with the underlying service via a REST or similar interface. In a production system, the only ports that should be exposed are those required for the operation of the system.

> **Warning:** Some of the services used in the example (for example, Postgres and Elasticsearch) persist data to disk. For this example, the volumes used for this persisted data are internal to the Docker container. This means that when the Docker network is torn down, all data is cleaned up along with the containers. In a production system, these volumes would be mounted onto permanent storage.

> **Warning:** Passwords are stored in plaintext in configuration files. In a production system, passwords should be extracted from a secure keystore at runtime.

> **Warning:** Network connections are not secured. In a production system, connections between services should be TLS-enabled, with a certificate authority (CA) provided.

> **Warning:** The memory use of the containers is only suitable for light demonstration loads. In a production setup, containers need to be given sufficient memory based on memory profiling.

> **Warning:** The versions of the Docker images used in the example may become outdated. In a production system, only the latest patched versions should be used.

Network Configuration

In this compose file, define the network that will be used to connect all the running containers:

Listing 30: etc/network-docker-compose.yml

```
# Create with `docker network create monitoring`
# Note that `external: false` will fail the docker-compose execution if the
↪network `monitoring` already exists
```

(continues on next page)

(continued from previous page)

```
version: "3.8"

networks:
  default:
    name: monitoring
    external: false
```

Postgres Setup

Using only a single Postgres container, create databases for the domain, along with Canton and index databases for each participant. To do this, mount *postgres-init.sql* into the Postgres-initialized directory. Note that in a production environment, passwords must not be inlined inside config.

Listing 31: etc/postgres-docker-compose.yml

```
services:
  postgres:
    image: postgres:14.8-bullseye
    hostname: postgres
    container_name: postgres
    environment:
      - POSTGRES_USER=pguser
      - POSTGRES_PASSWORD=pgpass
    volumes:
      - ../etc/postgres-init.sql:/docker-entrypoint-initdb.d/init.sql
    expose:
      - "5432"
    ports:
      - "5432:5432"
    healthcheck:
      test: "pg_isready -U postgres"
      interval: 5s
      timeout: 5s
      retries: 5
```

Listing 32: etc/postgres-init.sql

```
create database canton1db;
create database index1db;

create database domain0db;

create database canton2db;
create database index2db;
```

Domain Setup

Run the domain with the *-log-profile container* that writes plain text to standard out at debug level.

Listing 33: etc/domain0-docker-compose.yml

```
services:
  domain0:
    image: digitalasset/canton-open-source:2.5.1
    container_name: domain0
    hostname: domain0
    volumes:
      - ./domain0.conf:/canton/etc/domain0.conf
    command: daemon --log-profile container --config etc/domain0.conf
    expose:
      - "10018"
      - "10019"
      - "10020"
    ports:
      - "10018:10018"
      - "10019:10019"
      - "10020:10020"
```

Listing 34: etc/domain0.conf

```
canton {
  domains {
    domain0 {
      storage {
        type = postgres
        config {
          dataSourceClass = "org.postgresql.ds.PGSimpleDataSource"
          properties = {
            databaseName = "domain0db"
            serverName = "postgres"
            portNumber = "5432"
            user = pguser
            password = pgpass
          }
        }
      }
      public-api {
        port = 10018
```

(continues on next page)

(continued from previous page)

```
        address = "0.0.0.0"
      }
      admin-api {
        port = 10019
        address = "0.0.0.0"
      }
    }
  }
  monitoring.metrics.reporters = [{
    type = prometheus
    address = "0.0.0.0"
    port = 10020
  }]
}
```

Participant Setup

The participant container has two files mapped into it on container creation. The *.conf* file provides details of the domain and database locations. An HTTP metrics endpoint is exposed that returns metrics in the Prometheus Text Based Format. By default, participants do not connect to remote domains, so a bootstrap script is provided to accomplish that.

Listing 35: etc/participant1-docker-compose.yml

```
services:
  participant1:
    image: digitalasset/canton-open-source:2.5.1
    container_name: participant1
    hostname: participant1
    volumes:
      - ./participant1.conf:/canton/etc/participant1.conf
      - ./participant1.bootstrap:/canton/etc/participant1.bootstrap
    command: daemon --log-profile container --config etc/participant1.conf --
↪bootstrap etc/participant1.bootstrap
    expose:
      - "10011"
      - "10012"
      - "10013"
    ports:
      - "10011:10011"
      - "10012:10012"
      - "10013:10013"
```

Listing 36: etc/participant1.bootstrap

```
participant1.domains.connect(domain0.defaultDomainConnection)
```

Listing 37: etc/participant1.conf

```
canton {
  participants {
```

(continues on next page)

(continued from previous page)

```
    participant1 {
      storage {
        type = postgres
        config {
        dataSourceClass = "org.postgresql.ds.PGSimpleDataSource"
          properties = {
            databaseName = "canton1db"
            serverName = "postgres"
            portNumber = "5432"
            user = pguser
            password = pgpass
          }
        }
        ledger-api-jdbc-url = "jdbc:postgresql://postgres:5432/index1db?
↪user=pguser&password=pgpass"
      }
      ledger-api {
        port = 10011
        address = "0.0.0.0"
      }
      admin-api {
        port = 10012
        address = "0.0.0.0"
      }
    }
  }
  monitoring.metrics.reporters = [{
    type = prometheus
    address = "0.0.0.0"
    port = 10013
  }]
  remote-domains.domain0 {
    public-api {
      address="domain0"
      port = 10018
    }
    admin-api {
      address = "domain0"
      port = 10019
    }
  }
}
```

The setup for participant2 is identical, except that the name and ports are changed.

Listing 38: etc/participant2-docker-compose.yml

```
services:
  participant2:
    image: digitalasset/canton-open-source:2.5.1
    container_name: participant2
    hostname: participant2
    volumes:
      - ./participant2.conf:/canton/etc/participant2.conf
```

(continues on next page)

(continued from previous page)

```
      - ./participant2.bootstrap:/canton/etc/participant2.bootstrap
    command: daemon --log-profile container --config etc/participant2.conf --
↪bootstrap etc/participant2.bootstrap
    expose:
      - "10021"
      - "10022"
      - "10023"
    ports:
      - "10021:10021"
      - "10022:10022"
      - "10023:10023"
```

Listing 39: etc/participant2.bootstrap

```
participant1.domains.connect(domain0.defaultDomainConnection)
```

Listing 40: etc/participant2.conf

```
canton {
  participants {
    participant1 {
      storage {
        type = postgres
        config {
        dataSourceClass = "org.postgresql.ds.PGSimpleDataSource"
          properties = {
            databaseName = "canton1db"
            serverName = "postgres"
            portNumber = "5432"
            user = pguser
            password = pgpass
          }
        }
        ledger-api-jdbc-url = "jdbc:postgresql://postgres:5432/index1db?
↪user=pguser&password=pgpass"
      }
      ledger-api {
        port = 10011
        address = "0.0.0.0"
      }
      admin-api {
        port = 10012
        address = "0.0.0.0"
      }
    }
  }
  monitoring.metrics.reporters = [{
    type = prometheus
    address = "0.0.0.0"
    port = 10013
  }]
  remote-domains.domain0 {
    public-api {
      address="domain0"
      port = 10018
```

(continues on next page)

(continued from previous page)

```
    }
    admin-api {
      address = "domain0"
      port = 10019
    }
  }
}
```

Logstash

Docker containers can specify a log driver to automatically export log information from the container to an aggregating service. The example exports log information in GELF, using Logstash as the aggregation point for all GELF streams. You can use Logstash to feed many downstream logging data stores, including Elasticsearch, Loki, and Graylog.

Listing 41: etc/logstash-docker-compose.yml

```
services:
  logstash:
    image: docker.elastic.co/logstash/logstash:8.5.1
    hostname: logstash
    container_name: logstash
    expose:
      - 12201/udp
    volumes:
      - ./pipeline.yml:/usr/share/logstash/config/pipeline.yml
      - ./logstash.yml:/usr/share/logstash/config/logstash.yml
      - ./logstash.conf:/usr/share/logstash/pipeline/logstash.conf
    ports:
      - "12201:12201/udp"
```

Logstash reads the *pipeline.yml* to discover the locations of all pipelines.

Listing 42: etc/pipeline.yml

```
- pipeline.id: main
  path.config: "/usr/share/logstash/pipeline/logstash.conf"
```

The configured pipeline reads GELF-formatted input, then outputs it to an Elasticsearch index prefixed with *logs-* and postfixed with the date.

Listing 43: etc/logstash.conf

```
# Main logstash pipeline

input {
  gelf {
    use_udp => true
```

(continues on next page)

(continued from previous page)

```
    use_tcp => false
    port => 12201
  }
}

filter {}

output {

  elasticsearch {
    hosts => ["http://elasticsearch:9200"]
    index => "logs-%{+YYYY.MM.dd}"
  }

}
```

The default Logstash settings are used, with the HTTP port bound to all host IP addresses.

Listing 44: etc/logstash.yml

```
# For full set of descriptions see
# https://www.elastic.co/guide/en/logstash/current/logstash-settings-file.html

http.host: "0.0.0.0"
```

Elasticsearch

Elasticsearch supports running in a clustered configuration with built-in resiliency. The example runs only a single Elasticsearch node.

Listing 45: etc/elasticsearch-docker-compose.yml

```
services:
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:8.5.2
    container_name: elasticsearch
    environment:
      ELASTIC_PASSWORD: elastic
      node.name: elasticsearch
      cluster.name: elasticsearch
      cluster.initial_master_nodes: elasticsearch
      xpack.security.enabled: false
      bootstrap.memory_lock: true
    ulimits:
      memlock:
        soft: -1
        hard: -1
    expose:
      - 9200
    ports:
      - 9200:9200
```

(continues on next page)

(continued from previous page)

```
    healthcheck:
      test: "curl -s -I http://localhost:9200 | grep 'HTTP/1.1 200 OK'"
      interval: 10s
      timeout: 10s
      retries: 10
```

Kibana

Kibana provides a UI that allows the Elasticsearch log index to be searched.

Listing 46: etc/kibana-docker-compose.yml

```
services:
  kibana:
    image: docker.elastic.co/kibana/kibana:8.5.2
    container_name: kibana
    expose:
      - 5601
    ports:
      - 5601:5601
    environment:
      - SERVERNAME=kibana
      - ELASTICSEARCH_HOSTS=http://elasticsearch:9200
    healthcheck:
      test: "curl -s -I http://localhost:5601 | grep 'HTTP/1.1 302 Found'"
      interval: 10s
      timeout: 10s
      retries: 10
```

You must manually configure a data view to view logs. See *Kibana Log Monitoring* for instructions.

cAdvisor

cAdvisor exposes container system metrics (CPU, memory, disk, and network) to Prometheus. It also provides a UI to view these metrics.

Listing 47: etc/cadvisor-docker-compose.yml

```
services:
  cadvisor:
    image: gcr.io/cadvisor/cadvisor:v0.45.0
    container_name: cadvisor
    hostname: cadvisor
    privileged: true
    devices:
      - /dev/kmsg:/dev/kmsg
    volumes:
      - /var/run:/var/run:ro
```

(continues on next page)

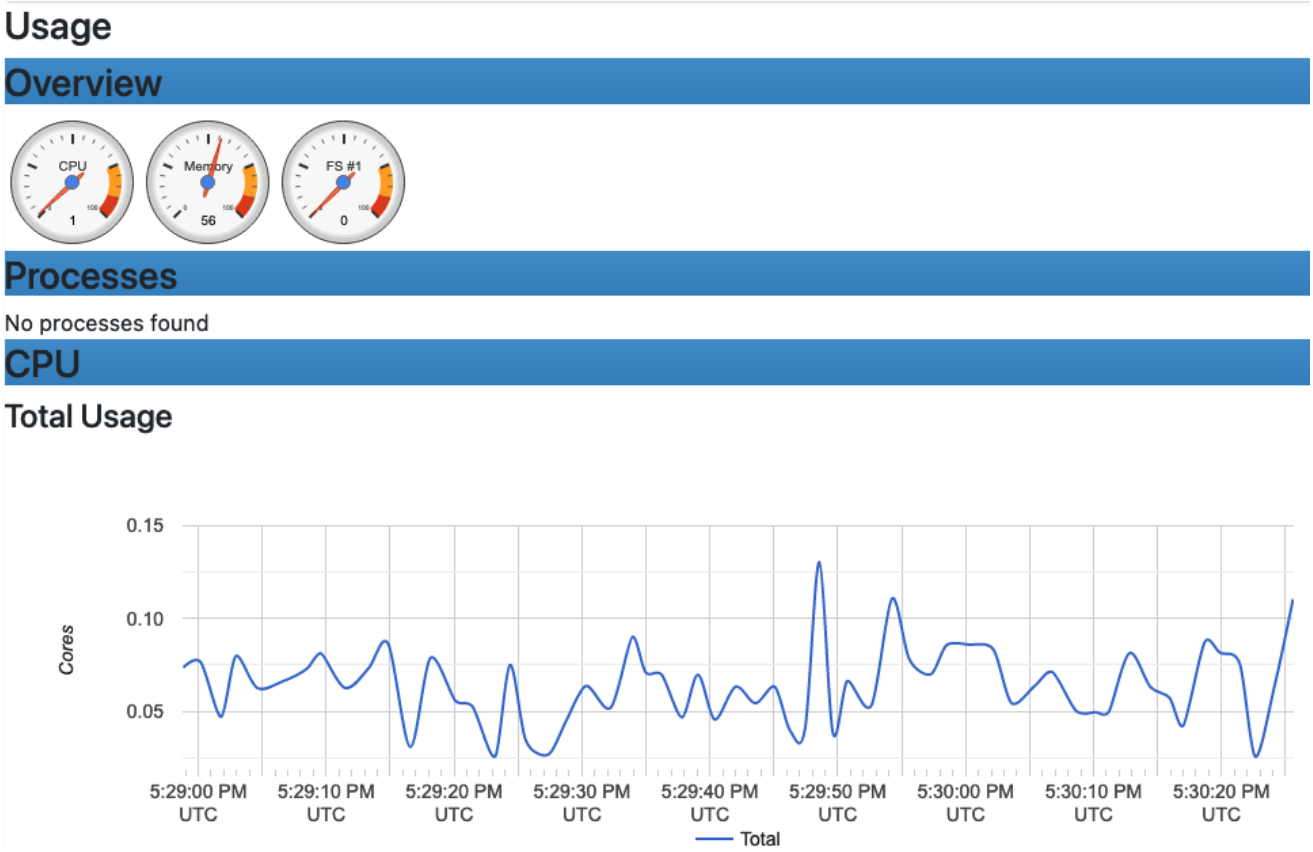
(continued from previous page)

```
      - /var/run/docker.sock:/var/run/docker.sock:ro
      # Although the following two directories are not present on OSX removing
↪them stops cAdvisor working
      # Maybe some internal logic checks for the existence of the directory.
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
    expose:
      - 8080
    ports:
      - "8080:8080"
```

To view container metrics:

1. Navigate to http://localhost:8080/docker/.
2. Select a Docker container of interest.

You should now see a UI similar to the one shown.



Prometheus-formatted metrics are available by default at http://localhost:8080/metrics.

Prometheus

Configure Prometheus with *prometheus.yml* to provide the endpoints from which metric data should be scraped. By default, port *9090* can query the stored metric data.

Listing 48: etc/prometheus-docker-compose.yml

```
services:
  prometheus:
    image: prom/prometheus:v2.40.6
    container_name: prometheus
    hostname: prometheus
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
    ports:
      - 9090:9090
```

Listing 49: etc/prometheus.yml

```
global:
  scrape_interval: 15s
  scrape_timeout: 10s
  evaluation_interval: 1m

scrape_configs:

  - job_name: canton
    static_configs:
      - targets:
        - domain0:10020
        - participant1:10013
        - participant2:10023

  - job_name: cadvisor
    static_configs:
      - targets:
        - cadvisor:8080

    # Exclude container labels by default
    # curl cadvisor:8080/metrics to see all available labels
    metric_relabel_configs:
      - regex: "container_label_.*"
        action: labeldrop
```

Grafana

Grafana is provided with:

The connection details for the Prometheus metric store
The username and password required to use the web UI
The location of any externally provided dashboards
The actual dashboards

Note that the *Metric Count* dashboard referenced in the docker-compose.yml file (*grafana-message-count-dashboard.json*) is not inlined below. The reason is that this is not hand-configured but built via the web UI and then exported. See *Grafana Metric Monitoring* for instructions to log into Grafana and display the dashboard.

Listing 50: etc/grafana-docker-compose.yml

```
services:
  grafana:
    image: grafana/grafana:9.3.1-ubuntu
    container_name: grafana
    hostname: grafana
    volumes:
      - ./grafana.ini:/etc/grafana/grafana.ini
      - ./grafana-datasources.yml:/etc/grafana/provisioning/datasources/default.
↪yml
      - ./grafana-dashboards.yml:/etc/grafana/provisioning/dashboards/default.yml
      - ./grafana-message-count-dashboard.json:/var/lib/grafana/dashboards/
↪grafana-message-count-dashboard.json
    ports:
      - 3000:3000
```

Listing 51: etc/grafana.ini

```
instance_name = "docker-compose"

[security]
admin_user = "grafana"
admin_password = "grafana"

[unified_alerting]
enabled = false

[alerting]
enabled = false

[plugins]
plugin_admin_enabled = true
```

Listing 52: etc/grafana-datasources.yml

```
---
apiVersion: 1

datasources:
- name: prometheus
```

(continues on next page)

(continued from previous page)

```
  type: prometheus
  access: proxy
  orgId: 1
  uid: prometheus
  url: http://prometheus:9090
  isDefault: true
  version: 1
  editable: false
```

Listing 53: etc/grafana-dashboards.yml

```
---
apiVersion: 1

providers:
  - name: local
    orgId: 1
    folder: ''
    folderUid: default
    type: file
    disableDeletion: true
    updateIntervalSeconds: 30
    allowUiUpdates: true
    options:
      path: /var/lib/grafana/dashboards
      foldersFromFilesStructure: true
```

Dependencies

There are startup dependencies between the Docker containers. For example, the domain needs to be running before the participant, and the database needs to run before the domain.

The *yaml* anchor *x-logging* enabled GELF container logging and is duplicated across the containers where you want to capture logging output. Note that the host address is the host machine, not a network address (on OSX).

Listing 54: etc/dependency-docker-compose.yml

```
x-logging: &logging
  driver: gelf
  options:
    # Should be able to use "udp://logstash:12201"
    gelf-address: "udp://host.docker.internal:12201"

services:

  logstash:
    depends_on:
      elasticsearch:
        condition: service_healthy

  postgres:
    logging:
```

(continues on next page)

(continued from previous page)

```
    <<: *logging
  depends_on:
    logstash:
      condition: service_started

domain0:
  logging:
    <<: *logging
  depends_on:
    postgres:
      condition: service_healthy
    logstash:
      condition: service_started

participant1:
  logging:
    <<: *logging
  depends_on:
    domain0:
      condition: service_started
    logstash:
      condition: service_started

participant2:
  logging:
    <<: *logging
  depends_on:
    domain0:
      condition: service_started
    logstash:
      condition: service_started

kibana:
  depends_on:
    elasticsearch:
      condition: service_healthy

grafana:
  depends_on:
    prometheus:
      condition: service_started
```

Docker Images

The Docker images need to be pulled down before starting the network:

digitalasset/canton-open-source:2.5.1
docker.elastic.co/elasticsearch/elasticsearch:8.5.2
docker.elastic.co/kibana/kibana:8.5.2
docker.elastic.co/logstash/logstash:8.5.1
gcr.io/cadvisor/cadvisor:v0.45.0
grafana/grafana:9.3.1-ubuntu
postgres:14.8-bullseye
prom/prometheus:v2.40.6

Running Docker Compose

Since running *docker compose* with all the compose files shown above creates a long command line, a helper script *dc.sh* is used.

A minimum of **12GB** of memory is recommended for Docker. To verify that Docker is not running short of memory, run *docker stats* and ensure the total *MEM%* is not too high.

Listing 55: dc.sh

```
#!/bin/bash

if [ $# -eq 0 ];then
    echo "Usage: $0 <docker compose command>"
    echo "Use '$0 up --force-recreate --renew-anon-volumes' to re-create network"
    exit 1
fi

set -x

docker compose \
    -p monitoring \
    -f etc/network-docker-compose.yml \
    -f etc/cadvisor-docker-compose.yml \
    -f etc/elasticsearch-docker-compose.yml \
    -f etc/logstash-docker-compose.yml \
    -f etc/postgres-docker-compose.yml \
    -f etc/domain0-docker-compose.yml \
    -f etc/participant1-docker-compose.yml \
    -f etc/participant2-docker-compose.yml \
    -f etc/kibana-docker-compose.yml \
    -f etc/prometheus-docker-compose.yml \
    -f etc/grafana-docker-compose.yml \
    -f etc/dependency-docker-compose.yml \
    $*
```

Useful commands

```
./dc.sh up -d       # Spins up the network and runs it in the background

./dc.sh ps          # Shows the running containers

./dc.sh stop        # Stops the containers

./dc.sh start       # Starts the containers

./dc.sh down        # Stops and tears down the network, removing any created
containers
```

1.29.10.2 Connecting to Nodes

To interact with the running network, a Canton console can be used with a remote configuration. For example:

```
bin/canton -c etc/remote-participant1.conf
```

Remote configurations

Listing 56: etc/remote-domain0.conf

```
canton.remote-domains.domain0 {
  admin-api {
    address="0.0.0.0"
    port="10019"
  }
  public-api {
    address="0.0.0.0"
    port="10018"
  }
}
```

Listing 57: etc/remote-participant1.conf

```
canton {

  features.enable-testing-commands = yes  // Needed for ledger-api

  remote-participants.participant1 {
    ledger-api {
      address="0.0.0.0"
      port="10011"
    }
    admin-api {
      address="0.0.0.0"
      port="10012"
    }
  }
}
```

Listing 58: etc/remote-participant2.conf

```
canton {

  features.enable-testing-commands = yes  // Needed for ledger-api

  remote-participants.participant2 {
    ledger-api {
      address="0.0.0.0"
      port="10021"
    }
    admin-api {
```

(continues on next page)

(continued from previous page)

```
        address="0.0.0.0"
        port="10022"
      }
    }

}
```

Getting Started

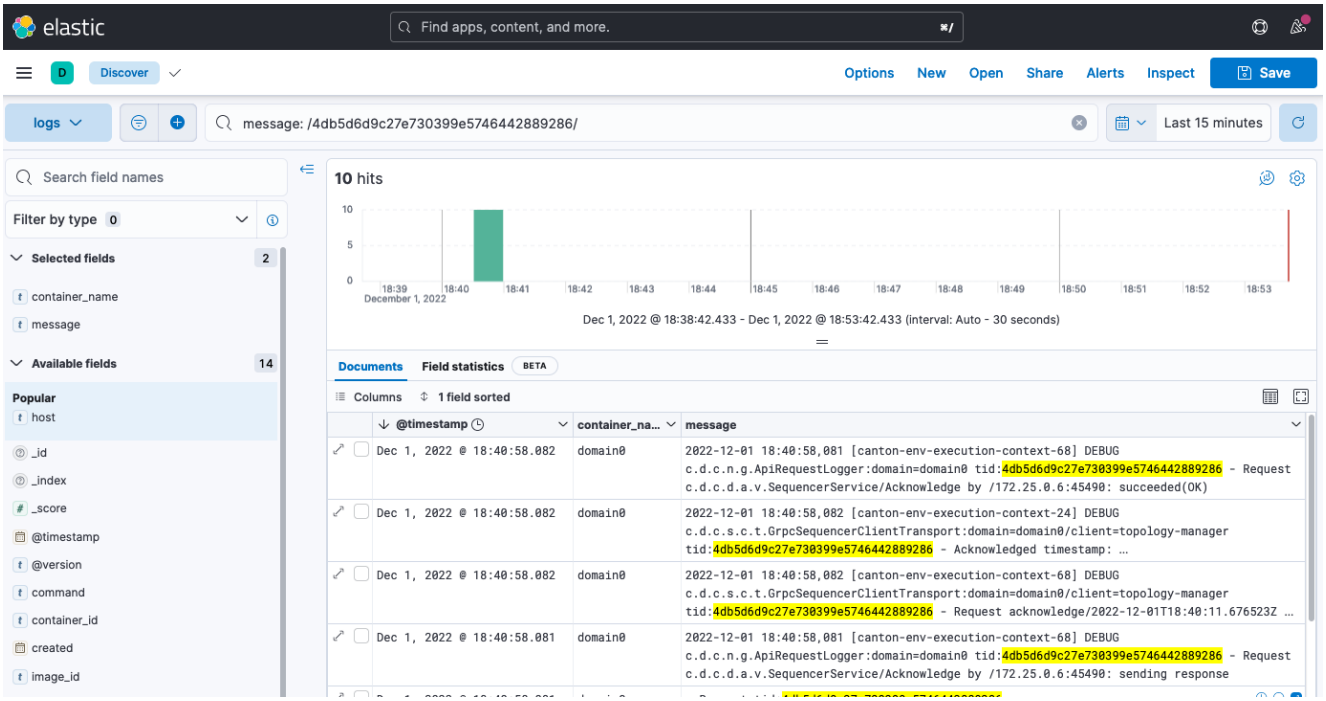
Using the previous scripts, you can follow the examples provided in the *Getting Started* guide.

1.29.10.3 Kibana Log Monitoring

When Kibana is started for the first time, you must set up a data view to allow view the log data:

1. Navigate to http://localhost:5601/.
2. Click **Explore on my own**.
3. From the menu select **Analytics** > **Discover**.
4. Click **Create data view**.
5. Save a data view with the following properties:
   Name: *Logs*
   Index pattern: *logs-**
   Timestamp field: *@timestamp*

You should now see a UI similar to the one shown here:



In the Kibana interface, you can:

Create a view based on selected fields
View log messages by logging timestamp
Filter by field value

Search for text
Query using either *KSQL* or *Lucene* query languages

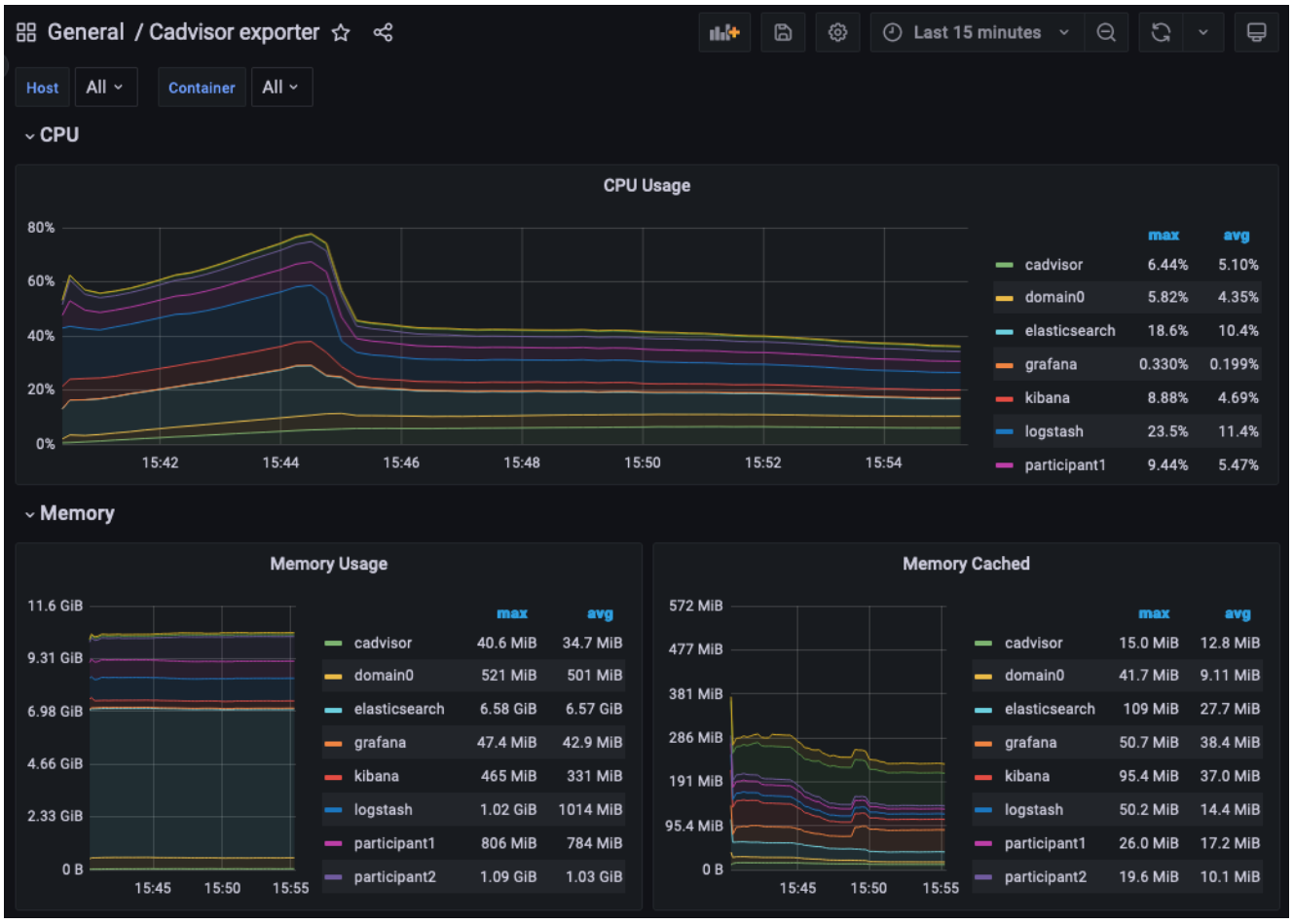
For more details, see the Kibana documentation. Note that querying based on plain text for a wide time window likely results in poor UI performance. See *Logging Improvements* for ideas to improve it.

1.29.10.4 Grafana Metric Monitoring

You can log into the Grafana UI and set up a dashboard. The example imports a GrafanaLabs community dashboard that has graphs for cAdvisor metrics. The cAdvisor Export dashboard imported below has an ID of **14282**.

1. Navigate to http://localhost:3000/login.
2. Enter the username/password: *grafana*/*grafana*.
3. In the side border, select **Dashboards** and then **Import**.
4. Enter the dashboard ID *14282* and click **Load**.
5. On the screen, select **Prometheus** as the data source and click **Import**.

You should see a container system metrics dashboard similar to the one shown here:



See the Grafana documentation for how to configure dashboards. For information about which metrics are available, see the Metrics documentation in the Monitoring section of this user manual.

1.29.10.5 Monitoring Choices

This section documents the reasoning behind the technology used in the example monitoring setup.

Use Docker Log Drivers

Reasons:

Most Docker containers can be configured to log all debug output to stdout.
Containers can be run as supplied.
No additional dockerfile layers need to be added to install and start log scrapers.
There is no need to worry about local file naming, log rotation, and so on.

Use GELF Docker Log Driver

Reasons:

It is shipped with Docker.
It has a decodable JSON payload.
It does not have the size limitations of syslog.
A UDP listener can be used to debug problems.

Use Logstash

Reasons:

It is a lightweight way to bridge the GELF output provided by the containers into Elasticsearch.
It has a simple conceptual model (pipelines consisting of input/filter/output plugins).
It has a large ecosystem of input/filter and output plugins.
It externalizes the logic for mapping container logging output to a structures/ECS format.
It can be run with *stdin*/*stdout* input/output plugins for use with testing.
It can be used to feed Elasticsearch, Loki, or Graylog.
It has support for the Elastic Common Schema (ECS) if needed.

Use Elasticsearch/Kibana

Reasons:

Using Logstash with Elasticsearch and Kibana, the ELK stack, is a mature way to set up a logging infrastructure.
Good defaults for these products allow a basic setup to be started with almost zero configuration.
The ELK setup acts as a good baseline as compared to other options such as Loki or Graylog.

Use Prometheus/Grafana

Reasons:

Prometheus defines and uses the OpenTelemetry reference file format.
Exposing metrics via an HTTP endpoint allows easy direct inspection of metric values.
The Prometheus approach of pulling metrics from the underlying system means that the running containers do not need infrastructure to store and push metric data.
Grafana works very well with Prometheus.

1.29.10.6 Logging Improvements

This version of the example only has the logging structure provided via GELF. It is possible to improve this by:

Extracting data from the underlying containers as a JSON stream.
Mapping fields in this JSON data onto the ECS so that the same name is used for commonly used field values (for example, log level).
Configuring Elasticsearch with a schema that allows certain fields to be quickly filtered (for example, log level).

1.29.11 Glossary

1.29.11.1 cAdvisor

Container Advisor (cAdvisor) provides an overview of CPU, memory, disk, and network utilization for each of the Docker containers. It works by querying the Docker Engine API to get these statistics for each container. This lets you avoid layering the containers with a utility to perform these functions.

https://github.com/google/cadvisor

1.29.11.2 Docker Log Driver

Docker containers can be configured with a log driver that allows log output to be exported from the Docker container. Using log drivers to export logging information makes running another process on the Docker container for this unnecessary.

https://docs.docker.com/config/containers/logging/configure/

1.29.11.3 Docker Plugins

A Docker plugin is a way to extend Docker (for example, by adding a log driver).

https://docs.docker.com/engine/extend/

1.29.11.4 ECS

The Elastic Common Schema (ECS) defines a naming convention for fields used in Elasticsearch. For example, use *@timestamp* for timestamp.

https://www.elastic.co/guide/en/ecs/current/ecs-field-reference.html

1.29.11.5 Elasticsearch

Elasticsearch is a technology that allows JSON documents to be stored, indexed, and searched in near real time. It can be configured as a cluster with built-in resiliency.

https://www.elastic.co/guide/en/elasticsearch/reference/8.5/index.html

1.29.11.6 ELK

The ELK stack is an established way to enable capturing, indexing, and displaying log data.

https://www.elastic.co/what-is/elk-stack

1.29.11.7 GELF

The Graylog extended logging format (GELF) improves on syslog logging by providing structured messages that are not size-limited. GELF is one of the built-in logging drivers supported by Docker. The message format is compressed JSON.

https://docs.graylog.org/docs/gelf

1.29.11.8 Grafana

Grafana provides a web UI that allows the construction of dashboards showing metric data. This data can be queried against a Prometheus metric store.

https://grafana.com/grafana/

1.29.11.9 Graylog

Unlike Elasticsearch, Graylog is not a general-purpose indexing, analytics, and search tool. It is designed specifically for log data. This provides a simpler, more focused option with better defaults for logging.

https://www.graylog.org/about/

1.29.11.10 Logstash

Logstash is a service that allows a series of pipelines to be configured that read, filter, and manipulate data before writing it out. It has support for a multitude of input, filter, and output types. The GELF input reader and Elasticsearch output writer are of particular interest.

https://www.elastic.co/guide/en/logstash/current/introduction.html

1.29.11.11 Loki

Loki is a log aggregation system designed to store and query logs from all your applications and infrastructure. It displays log information inside Grafana, allowing a single UI to be used for both metric data and logs.

https://grafana.com/oss/loki/

1.29.11.12 Loki Log Driver

The Loki log driver is a Loki client that allows log information to be shipped from a Docker log file, similar to other log drivers. The message format is gRPC protobuf.

https://grafana.com/docs/loki/latest/clients/docker-driver/

1.29.11.13 MinIO

AWS S3 Compatible Storage (used by Loki).

https://min.io/product/s3-compatibility

1.29.11.14 OpenTelemetry

OpenTelemetry is an organization that works to standardize observability (an umbrella term that includes logging, metrics, and tracing).

https://opentelemetry.io/

1.29.11.15 Prometheus

Prometheus can be configured to scrape metric data from many endpoints. This metric data can then be queried by metric visualization tools such as Grafana.

https://prometheus.io/

1.29.11.16 Syslog

Syslog is a standard for logging messages that has been around since the 1980s. Syslog is one of the built-in logging drivers supported by Docker.

https://en.wikipedia.org/wiki/Syslog

1.30 Identity Management

On-ledger identity management focuses on the distributed aspect of identities across Canton system entities, while user identity management focuses on individual participants managing access of their users to their ledger APIs.

Canton comes with a built in identity management system used to manage on-ledger identities. The technical details are explained in the *architecture section*, while this write up here is meant to give a high level explanation.

The identity management system is self-contained and built without a trusted central entity or pre-defined root certificate such that anyone can connect with anyone, without the need of some central approval and without the danger of losing self-sovereignty.

1.30.1 Introduction

1.30.1.1 What is a Canton Identity?

When two system entities such as a participant, domain topology manager, mediator or sequencer communicate with each other, they will use asymmetric cryptography to encrypt messages and sign message contents such that only the recipient can decrypt the content, verify the authenticity of the message, or prove its origin. Therefore, we need a method to uniquely identify the system entities and a way to associate encryption and signing keys with them.

On top of that, Canton uses the contract language Daml, which represents contract ownership and rights through parties. But parties are not primary members of the Canton synchronisation protocol. They are represented by participants and therefore we need to uniquely identify parties and relate them to participants, such that a participant can represent several parties (and in Canton, a party can be represented by several participants).

1.30.1.2 Unique Identifier

A Canton identity is built out of two components: a random string `X` and a fingerprint of a public key `N`. This combination, `(X,N)`, is called a *unique identifier* and is assumed to be globally unique by design. This unique identifier is used in Canton to refer to particular parties, participants or domain entities. A system entity (such as a party) is described by the combination of role (party, participant, mediator, sequencer, domain topology manager) and its unique identifier.

The system entities require knowledge about the keys which will be used for encryption and signing by the respective other entities. This knowledge is distributed and therefore, the system entities require a way to verify that a certain association of an entity with a key is correct and valid. This is the purpose of the fingerprint of a public key in the unique identifier, which is referred to as *Namespace*. And the secret key of the corresponding namespace acts as the *root of trust* for that particular namespace, as explained later.

1.30.1.3 Topology Transactions

In order to remain flexible and be able to change keys and cryptographic algorithms, we don't identify the entities using a single static key, but we need a way to dynamically associate participants or domain entities with keys and parties with participants. We do this through topology transactions.

A topology transaction establishes a certain association of a unique identifier with either a key or a relationship with another identifier. There are several different types of topology transactions. The most general one is the `OwnerToKeyMapping`, which as the name says, *associates a key with a unique identifier*. Such a topology transaction will inform all other system entities that a certain system entity is using a specific key for a specific purpose, such as participant *Alice* of namespace *12345..* is using the key identified through the fingerprint *AABBCCDDEE..* to sign messages.

Now, this poses two questions: who authorizes these transactions, and who distributes them?

For the authorization, we need to look at the second part of the unique identifier, the *Namespace*. A topology transaction that refers to a particular unique identifier operates on that namespace and we require that such a topology transaction is authorized by the corresponding secret key through a cryptographic signature of the serialised topology transaction. This authorization can be either direct, if it is signed by the secret key of the namespace, or indirect, if it is signed by a delegated key. In order to delegate the signing right to another key, there are other topology transactions of type *NamespaceDelegation* or *IdentifierDelegation* that allow one to do that. A *namespace delegation* delegates entire namespaces to a certain key, such as saying the key identifier through the fingerprint *AABBCCDDEE* is now allowed to authorize topology transactions within the namespace of the key *VVWWXXYYZZ* . An *identifier delegation* delegates authority over a certain identifier to a key, which means that the delegation key can only authorize topology transactions that act on a specific identifier and not the entire namespace.

Now, signing of topology transactions happens in a `TopologyManager`. Canton has many topology managers. In fact, every participant node and every domain have topology managers with exactly the same functional capabilities, just different impact. They can create new keys, new namespaces and the identity of new participants, parties and even domains. And they can export these topology transactions such that they can be imported at another topology manager. This allows to manage Canton identities in quite a wide range of ways. A participant can operate their own topology manager which allows them individually to manage their parties. Or they can associate themselves with another topology manager and let them manage the parties that they represent or keys they use. Or something in between, depending on the introduced delegations and associations.

The difference between the domain topology manager and the participant topology manager is that the domain topology manager establishes the valid topology state in a particular domain by distributing topology transactions in a way that every domain member ends up with the same topology state. However, the domain topology manager is just a gate keeper of the domain that decides who is let in and who not on that particular domain, but the actual topology statements originate from various sources. As such, the domain topology manager can only block the distribution, but cannot fake topology transactions.

The participant topology manager only manages an isolated topology state. However, there is a dispatcher attached to this particular topology manager that attempts to register locally registered identities with remote domains, by sending them to the domain topology managers, who then decide on whether they want to include them or not.

The careful reader will have noted that the described identity system indeed does not have a single root of trust or decision maker on who is part of the overall system or not. But also that the topology state for the distributed synchronisation varies from domain to domain, allowing very flexible topologies and setups.

1.30.1.4 Legal Identities

In Canton, we separate a system identity from the legal identity. While the above mechanism allows to establish a common, verified and authorized knowledge of system entities, it doesn't guarantee that a certain unique identifier really corresponds to a particular legal identity. Even more so, while the unique identifier remains stable, a legal identity might change, for example in the case of a merger of two companies. Therefore, Canton provides an administrative command which allows to associate a randomized system identity with a human readable *display name* using the `participant.parties.set_display_name` command.

Note: A party display name is private to the participant. If such names should be shared among participants, we recommend to build a corresponding Daml workflow and some automation logic, listening to the results of the Daml workflow and updating the display name accordingly.

1.30.1.5 Life of a Party

In the tutorials, we use the `participant.parties.enable("name")` function to setup a party on a participant. To understand the identity management system in Canton, it helps to look at the steps under the hood of how a new party is added:

1. The `participant.parties.enable` function determines the unique identifier of the participant: `participant.id`.
2. The party name is built as `name::<namespace>`, where the `namespace` is the one of the participant.
3. A new party to participant mapping is authorized on the Admin Api: `participant.topology.party_to_participant_mappings.authorize(...)`
4. The `ParticipantTopologyManager` gets invoked by the gRPC request, creating a new `SignedTopologyTransaction` and tests whether the authorization can be added to the local topology state. If it can, the new topology transaction is added to the store.
5. The `ParticipantTopologyDispatcher` picks up the new transaction and requests the addition on all domains via the `RegisterTopologyTransactionRequest` message sent to the topology manager through the sequencer.
6. A domain receives this request and processes it according to the policy (open or permissioned). The default setting is open.
7. If approved, the request service attempts to add the new topology transaction to the `DomainTopologyManager`.
8. The `DomainTopologyManager` checks whether the new topology transaction can be added to the domain topology state. If yes, it gets written to the local topology store.
9. The `DomainTopologyDispatcher` picks up the new transaction and sends it to all participants (and back to itself) through the sequencer.
10. The sequencer timestamps the transaction and embeds it into the transaction stream.
11. The participants receive the transaction, verify the integrity and correctness against the topology state and add it to the state with the timestamp of the sequencer, such that everyone has a synchronous topology state.

Note that the `participant.parties.enable` macro only works if the participant controls their namespace themselves, either directly by having the namespace key or through delegation (via `NamespaceDelegation`).

1.30.1.6 Participant Onboarding

Key to support topological flexibility is that participants can easily be added to new domains. Therefore, the on-boarding of new participants to domains needs to be secure but convenient. Looking at the console command, we note that in most examples, we are using the `connect` command to connect a participant to a domain. The connect command just wraps a set of admin-api commands:

```
val certificates = OptionUtil.emptyStringAsNone(certificatesPath).map { path =>
  BinaryFileUtil.readByteStringFromFile(path) match {
    case Left(err) => throw new IllegalArgumentException(s"failed to load ${path}
↪: ${err}")
    case Right(bs) => bs
  }
}
DomainConnectionConfig.grpc(
  SequencerAlias.Default,
  domainAlias,
  connection,
  manualConnect,
  domainId,
  certificates,
  priority,
  initialRetryDelay,
  maxRetryDelay,
  timeTrackerConfig,
)
```

```
// register the domain configuration
register(config.copy(manualConnect = true))
if (!config.manualConnect) {
  // fetch and confirm domain agreement
  if (config.sequencerConnections.nonBftSetup) { // agreement is removed with the
↪introduction of BFT domain.
    confirm_agreement(config.domain.unwrap)
  }
  reconnect(config.domain.unwrap, retry = false).discard
  // now update the domain settings to auto-connect
  modify(config.domain.unwrap, _.copy(manualConnect = false))
}
```

We note that from a user perspective, all that needs to happen by default is to provide the connection information and accepting the terms of service (if required by the domain) to set up a new domain connection. There is no separate on-boarding step performed, no giant certificate signing exercise happens, everything is set up during the first connection attempt. However, quite a few steps happen behind the scenes. Therefore, we briefly summarise the process here step by step:

1. The administrator of an existing participant needs to invoke the `domains.register` command to add a new domain. The mandatory arguments are a domain *alias* (used internally to refer to a particular connection) and the sequencer connection URL (`http` or `https`) including an optional port *http[s]://hostname[:port]/path*. Optional are a certificates path for a custom TLS certificate chain (otherwise the default jre root certificates are used) and the *domain id* of a domain. The *domain id* is the unique identifier of the domain that can be defined to prevent man-in-the-middle attacks (very similar to an ssh key fingerprint).
2. The participant opens a gRPC channel to the `SequencerConnectService`.
3. The participant contacts the `SequencerConnectService` and checks if using the domain

requires signing specific terms of services. If required, the terms of service are displayed to the user and an approval is locally stored at the participant for later. If approved, the participant attempts to connect to the sequencer.

4. The participant verifies that the remote domain is running a protocol version compatible with the participant's version using the `SequencerConnectService.handshake`. If the participant runs an incompatible protocol version, the connection will fail.
5. The participant will download and verify the domain id from the domain. The *domain id* can be used to verify the correct authorization of the topology transactions of the domain entities. If the domain id has been provided previously during the `domains.register` call (or in a previous session), the two ids will be compared. If they are not equal, the connection will fail. If the domain id was not provided during the `domains.register` call, the participant will use and store the one downloaded. We assume here that the domain id is obtained by the participant through a secure channel such that it is sure to be talking to the right domain. Therefore, this secure channel can be either something happening outside of Canton or can be provided by TLS during the first time we contact a domain.
6. The participant downloads the *static domain parameters*, which are the parameters used for the transaction protocol on the particular domain, such as the cryptographic keys supported on this domain.
7. The participant connects to the sequencer initially as an unauthenticated member. Such members can only send transactions to the domain topology manager. The participant then sends an initial set of topology transactions required to identify the participant and define the keys used by the participant to the `DomainTopologyManagerRequestService`. The request service inspects the validity of the transactions and decides based on the configured domain on-boarding policy. The currently supported policies are `open` (default) and `permissioned`. While `open` is convenient for permissionless systems and for development, it will accept any new participant and any topology transaction. The `permissioned` policy will accept the participant's onboarding transactions only if the participant has been added to the allow-list beforehand.
8. The request service forwards the transactions to the domain topology manager, who attempts to add it to the state (and thus trigger the distribution to the other members on a domain). The result of the onboarding request is sent to the unauthenticated member who disconnects upon receiving the response.
9. If the onboarding request is approved, the participant now attempts to connect to the sequencer as the actual participant.
10. Once the participant is properly enabled on the domain and its signing key is known, the participant can subscribe to the `SequencerService` with its identity. In order to do that and in order to verify the authorisation of any action on the `SequencerService`, the participant requires to obtain an authorization token from the domain. For this purpose, the participant requests a `Challenge` from the domain. The domain will provide it with a `nonce` and the fingerprint of the key to be used for authentication. The participant signs this nonce (together with the domain id) using the corresponding private key. The reason for the fingerprint is simple: the participant needs to sign the token using the participants signing key as defined by the domain topology state. However, as the participant will learn the true domain topology state only by reading from the `SequencerService`, it cannot know what the key is. Therefore, the domain discloses this part of the domain topology state as part of the authorisation challenge.
11. Using the created authentication token, the participant starts to use the *SequencerService*. On the domain side, the domain verifies the authenticity and validity of the token by verifying that the token is the expected one and is signed by the participant's signing key. The token is used to authenticate every gRPC invocation and needs to be renewed regularly.
12. The participant sets up the `ParticipantTopologyDispatcher`, which is the process that tries to push all topology transactions created at the participant node's topology manager to

the domain topology manager. If the participant is using its topology manager to manage its identity on its own, these transactions contain all the information about the registered parties or supported packages.

13. As mentioned above, the first set of messages received by the participant through the sequencer will contain the domain topology state, which includes the signing keys of the domain entities. These messages are signed by the sequencer and topology manager and are self-consistent. If the participants know the domain id, they can verify that they are talking to the expected domain and that the keys of the domain entities have been authorized by the owner of the key governing the domain id.
14. Once the initial topology transactions have been read, the participant is ready to process transactions and send commands.
15. When a participant is (re-)enabled, the domain topology dispatcher analyses the set of topology transactions the participant has missed before. It sends these transactions to the participant via the sequencer, before publicly enabling the participant. Therefore, when the participant starts to read messages from the sequencer, the initially received messages will be the topology state of the domain.

1.30.1.7 Default Initialization

The default initialization behaviour of participant and domain nodes is to run their own topology manager. This provides a convenient, automatic way to configure the nodes and make them usable without manual intervention, but it can be turned off by setting the `auto-init = false` configuration option **before** the first startup.

During the auto initialization, the following steps will happen:

1. On the domain, we generate four signing keys: one for the namespace and one each for the sequencer, mediator and topology manager. On the participant, we generate three keys: a namespace key, a signing key and an encryption key.
2. Using the fingerprint of the namespace, we generate the participant identity. For understandability, we use the node name used in the configuration file. This will change into a random identifier for privacy reasons. Once we've generated it, we set it using the `set_id` admin-api call.
3. We create a root certificate as `NamespaceDelegation` using the namespace key, signing with the namespace key.
4. Then, we create an `OwnerToKeyMapping` for the participant or domain entities.

The *init.identity* object can be set to control the behavior of the auto initialization. For instance, it is possible to control the identifier name that will be given to the node during the initialization. There are 3 possible configurations:

1. Use the node name as the node identifier

```
canton.participants.participant1.init.identity.node-identifier.type = config
```

2. Explicitly set a name

```
canton.participants.participant1.init.identity.node-identifier.type = explicit
canton.participants.participant1.init.identity.node-identifier.name = MyName
```

3. Generate a random name

```
canton.participants.participant1.init.identity.node-identifier.type = random
```

1.30.1.8 Identity Setup Guide

As explained, Canton nodes auto-initialise themselves by default, running their own topology managers. This is convenient for development and prototyping. Actual deployments require more care and therefore, this section should serve as a brief guideline.

Canton topology managers have one crucial task they must not fail at: do not lose access to or control of the root of trust (namespace keys). Any other key problem can somehow be recovered by revoking an old key and issuing a new owner to key association. Therefore, it is advisable that participants and parties are associated with a namespace managed by a topology manager that has sufficient operational setups to guarantee the security and integrity of the namespace.

Therefore, a participant or domain can

1. Run their own topology manager with their identity namespace key as part of the participant node.
2. Run their own topology manager on a detached computer in a self-built setup that exports topology transactions and transports them to the respective node (i.e. via burned CD roms).
3. Ask a trusted topology manager to issue a set of identifiers within the trusted topology manager's namespace as delegations and import the delegations to the local participant topology manager.
4. Let a trusted topology manager manage all the topology state on-behalf.

Obviously, there are more combinations and options possible, but these options here describe some common options with different security and recoverability options.

In order to reduce the risk of losing namespace keys, additional keys can be created and allowed to operate on a certain namespace. In fact, we recommend doing this and avoid storing the root key on a live node.

1.30.2 User Identity Management

So far we have covered how on-ledger identities are managed.

Every participant also needs to manage access to their local Ledger API and be able to give applications permission to read or write to that API on behalf of parties. While an on-ledger identity is represented as a party, an application on the Ledger API is represented and managed as a user. A ledger API server manages applications' identities through:

authentication: recognizing which user an application corresponds to (essentially by matching an application name with a user name)
authorization: knowing which rights an authenticated user has and restricting their Ledger API access according to those rights

Authentication is based on JWT and covered in the application development / authorization section of the manual; the related Ledger API authorization configuration is covered in the *Ledger API JWT configuration section*.

Authorization is managed by the Ledger API's User Management Service. In essence, a user is a mapping from a user name to a set of parties with read or write permissions. In more detail a user consists of:

a user id (also called user name)
an active/deactivated status (can be used to temporarily ban a user from accessing the Ledger API)
an optional primary party (indicates which party to use by default when submitting a Ledger API command requests as this user)
a set of user rights (describes whether a user has access to the admin portion of the Ledger API and what parties this user can act or read as)
a set of custom annotations (string based key-value pairs, stored locally on the Ledger API server, that can be used to attach extra information to this party, e.g. how it relates to some business entity)

All these properties except the user id can be modified. To learn more about annotations refer to the Ledger API Reference documentation . For an overview of the ledger API's UserManagementService, see this section.

You can manage users through the *Canton console user management commands*, an alpha feature. See the cookbook below for some concrete examples of how to manage users.

1.30.3 Cookbook

1.30.3.1 Manage Users

In this section, we present how you can manage participant users using the Canton console commands. First, we create three parties that we'll use in subsequent examples:

```
@ val Seq(alice, bob, eve) = Seq("alice", "bob", "eve").map(p => participant1.
↪parties.enable(name = p, waitForDomain = DomainChoice.All))
Seq(alice, bob, eve) : Seq[PartyId] = List(alice::12207af325a3...,
↪bob::12207af325a3..., eve::12207af325a3...)
```

Create

Next, create a user called `myuser` with act-as `alice` and read-as `bob` permissions and active user status. This user's primary party is `alice`. The user is not an administrator and has some custom annotations.

```
@ val user = participant1.ledger_api.users.create(id = "myuser", actAs =
↪Set(alice), readAs = Set(bob), primaryParty = Some(alice), participantAdmin =
↪false, isActive = true, annotations = Map("foo" -> "bar", "description" ->
↪"This is a description"))
user : User = User(
  id = "myuser",
  primaryParty = Some(value = alice::12207af325a3...),
  isActive = true,
  annotations = Map("foo" -> "bar", "description" -> "This is a description"),
  identityProviderId = ""
)
```

There are some restrictions for what constitutes a valid annotation key. In contrast, the only constraint for annotation values is that they must not be empty. To learn more about annotations refer to the Ledger API Reference documentation.

Update

You can update a user's primary party, active/deactivated status and annotations. (You can also change what rights a user has, but using a different method presented further below.)

In the following snippet, you change the user's primary party to be unassigned, leave the active/deactivated status intact, and update the annotations. In the annotations, you change the value of the `description` key, remove the `foo` key and add the new `baz` key. The return value contains the updated state of the user:

```
@ val updatedUser = participant1.ledger_api.users.update(id = user.id, modifier =
↪user => { user.copy(primaryParty = None, annotations = user.annotations.updated(
↪"description", "This is a new description").removed("foo").updated("baz", "bar
↪")) })
updatedUser : User = User(
  id = "myuser",
  primaryParty = None,
  isActive = true,
  annotations = Map("baz" -> "bar", "description" -> "This is a new description"),
  identityProviderId = ""
)
```

You can also update the user's identity provider id. In the following snippets, you change the user's identity provider id to the newly created one. Note that originally the user belonged to the default identity provider whose id is represented as the empty string `""`.

```
@ participant1.ledger_api.identity_provider_config.create("idp-id1",
↪isDeactivated = false, jwksUrl = "http://someurl", issuer = "issuer1", audience
↪= None)
res4: com.digitalasset.canton.ledger.api.domain.IdentityProviderConfig =
↪IdentityProviderConfig(
  identityProviderId = Id(value = "idp-id1"),
  isDeactivated = false,
  jwksUrl = JwksUrl(value = "http://someurl"),
  issuer = "issuer1",
  audience = None
)
```

```
@ participant1.ledger_api.users.update_idp("myuser", sourceIdentityProviderId="",
↪targetIdentityProviderId="idp-id1")
```

```
@ participant1.ledger_api.users.get("myuser", identityProviderId="idp-id1")
res6: User = User(
  id = "myuser",
  primaryParty = None,
  isActive = true,
  annotations = Map("baz" -> "bar", "description" -> "This is a new description"),
  identityProviderId = "idp-id1"
)
```

You can change the user's identity provider id back to the default one:

```
@ participant1.ledger_api.users.update_idp("myuser", sourceIdentityProviderId=
↪"idp-id1", targetIdentityProviderId="")
```

```
@ participant1.ledger_api.users.get("myuser", identityProviderId="")
res8: User = User(
  id = "myuser",
  primaryParty = None,
  isActive = true,
  annotations = Map("baz" -> "bar", "description" -> "This is a new description"),
  identityProviderId = ""
)
```

Inspect

You can fetch the current state of the user as follows:

```
@ participant1.ledger_api.users.get(user.id)
res9: User = User(
  id = "myuser",
  primaryParty = None,
  isActive = true,
  annotations = Map("baz" -> "bar", "description" -> "This is a new description"),
  identityProviderId = ""
)
```

You can query what rights a user has:

```
@ participant1.ledger_api.users.rights.list(user.id)
res10: UserRights = UserRights(
  actAs = Set(alice::12207af325a3...),
  readAs = Set(bob::12207af325a3...),
  participantAdmin = false,
  identityProviderAdmin = false
)
```

You can grant more rights. The returned value contains only newly granted rights; it does not contain rights the user already had even if you attempted to grant them again (like the read-as `alice` right in this example):

```
@ participant1.ledger_api.users.rights.grant(id = user.id, actAs = Set(alice,
↪bob), readAs = Set(eve), participantAdmin = true)
res11: UserRights = UserRights(
  actAs = Set(bob::12207af325a3...),
  readAs = Set(eve::12207af325a3...),
  participantAdmin = true,
  identityProviderAdmin = false
)
```

You can revoke rights from the user. Again, the returned value contains only rights that were actually removed:

```
@ participant1.ledger_api.users.rights.revoke(id = user.id, actAs = Set(bob),
↪readAs = Set(alice), participantAdmin = true)
res12: UserRights = UserRights(
  actAs = Set(bob::12207af325a3...),
  readAs = Set(),
  participantAdmin = true,
```

(continues on next page)

(continued from previous page)

```
  identityProviderAdmin = false
)
```

Now that you have granted and revoked some rights, you can fetch all of the user's rights again and see what they are:

```
@ participant1.ledger_api.users.rights.list(user.id)
res13: UserRights = UserRights(
  actAs = Set(alice::12207af325a3...),
  readAs = Set(bob::12207af325a3..., eve::12207af325a3...),
  participantAdmin = false,
  identityProviderAdmin = false
)
```

Also, multiple users can be fetched at the same time. In order to do that, first create another user called `myotheruser` and then list all the users whose user name starts with `my`:

```
@ participant1.ledger_api.users.create(id = "myotheruser")
res14: User = User(
  id = "myotheruser",
  primaryParty = None,
  isActive = true,
  annotations = Map(),
  identityProviderId = ""
)
```

```
@ participant1.ledger_api.users.list(filterUser = "my")
res15: UsersPage = UsersPage(
  users = Vector(
    User(
      id = "myotheruser",
      primaryParty = None,
      isActive = true,
      annotations = Map(),
      identityProviderId = ""
    ),
    User(
      id = "myuser",
      primaryParty = None,
      isActive = true,
      annotations = Map("baz" -> "bar", "description" -> "This is a new
↪description"),
      identityProviderId = ""
    )
  ),
  nextPageToken = ""
)
```

Decommission

You can delete a user by its id:

```
@ participant1.ledger_api.users.delete("myotheruser")
```

You can confirm it has been removed by e.g. listing it:

```
@ participant1.ledger_api.users.list("myotheruser")
res17: UsersPage = UsersPage(users = Vector(), nextPageToken = "")
```

If you want to prevent a user from accessing the ledger API it may be better to deactivate it rather than deleting it. A deleted user can be recreated as if it never existed in the first place, while a deactivated user must be explicitly reactivated to be able to access the ledger API again.

```
@ participant1.ledger_api.users.update("myuser", user => user.copy(isActive =
↪false))
res18: User = User(
  id = "myuser",
  primaryParty = None,
  isActive = false,
  annotations = Map("baz" -> "bar", "description" -> "This is a new description"),
  identityProviderId = ""
)
```

Configure a default Participant Admin

Fresh participant nodes come with a default participant admin user called `participant_admin`, which can be used to bootstrap other users. You might prefer to have an admin user with a different user id ready on a participant startup. For such situations, you can specify an additional participant admin user with the user id of your choice.

Note: If a user with the specified id already exists, then no additional user will be created, even if the preexisting user was not an admin user.

Listing 59: additional-admin.conf

```
canton.participants.myparticipant.ledger-api.user-management-service.additional-
↪admin-user-id = "my-admin-id"
```

1.30.3.2 Adding a new Party to a Participant

The simplest operation is adding a new party to a participant. For this, we add it normally at the topology manager of the participant, which in the default case is part of the participant node. There is a simple macro to enable the party on a given participant if the participant is running their own topology manager:

```
val name = "Gottlieb"
participant1.parties.enable(name)
```

This will create a new party in the namespace of the participants topology manager.

And there is the corresponding disable macro:

```
participant1.parties.disable(name)
```

The macros themselves just use `topology.party_to_participant_mappings.authorize` to create the new party, but add some convenience such as automatically determining the parameters for the `authorize` call.

Note: Please note that the `participant.parties.enable` macro will add the parties to the same namespace as the participant is in. It only works if the participant has authority over that namespace either by possessing the root or a delegated key.

Important: This feature is only available in Canton Enterprise

---

1.30.3.3 Migrate Party to Another Participant Node

Parties are only weakly tied to participant nodes. They can be allocated in their own namespace and then be delegated to a given participant. For simplicity and convenience, the participant creates new parties in their own namespace by default.

The weak coupling of parties to participants allows you to migrate parties together with their active contract set from one participant node to another. Note, the process below works only for parties that are hosted on a single node. Also, if the party is not fully controlled by the source participant node, you need to prepare the topology state change appropriately, disabling the party on the source node and delegating the party to the target node.

Note: Please note that the entire system needs to be totally quiet for this process to succeed. You currently can not migrate a party under load. If you migrate a party on a system that processes transactions, the processing data will eventually become corrupt breaking your node.

Turn off transaction processing on the domain by setting the rate to 0 and wait for all timeouts to have elapsed (mediator & participant reaction timeout):

```
mydomain.service.set_max_rate_per_participant(0)
// wait until mediator + participant reaction timeouts elapsed!
```

Starting with a party Alice being allocated on participant1:

```
@ val alice = participant1.parties.enable("Alice")
alice : PartyId = Alice::12204dc1e4c4...
```

To migrate Alice to participant2, we follow a four-step process. First, we need to obtain the target participant id. In this example, we read it from the participant id:

```
@ val targetParticipantId = participant2.id
targetParticipantId : ParticipantId = PAR::participant2::12207334a68d...
```

Next, we deactivate the party on the origin participant and store the party's active contract set in a file by using the repair macros which are part of the enterprise edition:

```
@ repair.party_migration.step1_hold_and_store_acs(alice, participant1,
↪targetParticipantId, "alice.acs.gz")
res3: Map[DomainId, Long] = Map()
```

The last argument is the name of a file which the active contract set is stored as base64 encoded strings, ordered by domain-id and contract-id. This file then needs to be transferred offline to the target participant. Additionally, the repair macro will disable the party on the first participant. This is important in order to avoid breaking the consistency of the exported active contract set.

The target participant must then be disconnected from the domain before it can import data:

```
@ participant2.domains.disconnect("mydomain")
```

Once the domain is disconnected, invoke the import command:

```
@ repair.party_migration.step2_import_acs(participant2, "alice.acs.gz")
```

When importing is finished, reconnect to the domain using:

```
@ participant2.domains.reconnect("mydomain")
res6: Boolean = true
```

The last step on the target participant enables the party:

```
@ repair.party_migration.step3_enable_on_target(alice, participant2)
```

Finally, purge the active contract set on the origin participant:

```
@ participant1.domains.disconnect("mydomain")
```

```
@ repair.party_migration.step4_clean_up_source(alice, participant1, "alice.acs.gz
↪")
```

The above commands require interactive access to the participants and are supported as an alpha implementation. They work for parties that were allocated using standard methods on a single participant node. Otherwise, a few more manual steps are required to properly prepare the topology state before exporting and importing the topology state.

1.30.3.4 Party on Two Nodes

Note: this is an alpha feature only and is not supported in production.

Assuming we have party `("Alice", N1)` which we want to host on two participants: `("participant1", N1)` and `("participant2", N2)`. In this case, we have the party Alice in namespace `N1`, whereas the participant2 is in namespace `N2`. In order to set this up, we need to appropriately authorize the participants to act on behalf of the party and we need to correctly copy the active contract set.

Starting with a party being allocated on participant1:

```
@ val alice = participant1.parties.enable("Alice")
alice : PartyId = Alice::1220df7d96ce...
```

To add this party to participant2, participant2 must first agree to host the party. This is done by authorizing the `RequestSide.To` of the party to participant mapping on the target participant:

```
@ participant2.topology.party_to_participant_mappings.authorize(TopologyChangeOp.
↪Add, alice, participant2.id, RequestSide.To, ParticipantPermission.Submission)
res2: com.google.protobuf.ByteString = <ByteString@f166652 size=554 contents="\n\
↪247\004\n\327\001\n\322\001\n\317\001\022 065F3gyr8JKybE1s2NaUZw37lRbPTwaw2...">
```

You can restrict the permission of the node by setting the appropriate `ParticipantPermission` in the authorization call to either `Observation` or `Confirmation` instead of the default `Submission`. This allows setups where a party is hosted with `Submission` permissions on one node and `Confirmation` on another to increase the liveness of the system.

Note: The distinction between `Submission` and `Confirmation` is only enforced in the participant node. A malicious participant node with `Confirmation` permission for a certain party can submit transactions in the name of the party. This is due to Canton's high level of privacy where validators do not know the identity of the submitting participant. Therefore, a party who delegates `Confirmation` permissions to a participant should trust the participant sufficiently.

Before we continue, we need to ensure that the target participant is now disconnected from the affected domains, in order to avoid the target participant receiving transactions for the new party prior to the complete transfer of the active contract store. Therefore, we disconnect the participant from all domains:

```
@ participant2.domains.disconnect_all()
```

This is currently the reason why this feature is only supported as alpha: we can not guarantee that a user does not damage their system by accident due to forgetting to disconnect from the domain.

Next, add the `RequestSide.From` transaction such that the party is activated on the target participant:

```
@ participant1.topology.party_to_participant_mappings.authorize(TopologyChangeOp.
↪Add, alice, participant2.id, RequestSide.From, ParticipantPermission.Submission)
res4: com.google.protobuf.ByteString = <ByteString@56f89b1b size=556 contents="\n\
↪251\004\n\327\001\n\322\001\n\317\001\022 LJCGYk1KGUI0bPaSEJHCulUglzsQ3lIr2...">
```

Check that the party is now hosted by two participants:

```
@ participant1.parties.list("Alice")
res5: Seq[ListPartiesResult] = Vector(
  ListPartiesResult(
    party = Alice::1220df7d96ce...,
    participants = Vector(
      ParticipantDomains(
        participant = PAR::participant2::1220a532b115...,
        domains = Vector(
          DomainPermission(domain = mydomain::1220a35ef9f4..., permission =
↪Submission)
        )
      ),
      ParticipantDomains(
        participant = PAR::participant1::1220df7d96ce...,
        domains = Vector(
```

(continues on next page)

(continued from previous page)

```
          DomainPermission(domain = mydomain::1220a35ef9f4..., permission =
↪Submission)
        )
      )
    )
  )
)
```

In the next step, you store the active contract set of the party into a file.

Make sure that the participant to supply the ACS has seen some transactions after the topology state has become active, but those transactions should not involve the migrated party. So just run a health check:

```
@ participant1.health.ping(participant1.id)
res6: Duration = 635 milliseconds
```

If there is no traffic on the participant node and you can be sure that nothing has changed for the party, you can just straight use the `repair.download` command. Otherwise, you must find the timestamp when the party was activated. One way to find that timestamp is by looking at the topology store of that particular domain connection:

```
@ val timestamp = participant1.topology.party_to_participant_mappings.
↪list(filterStore="mydomain", filterParty="Alice").map(_.context.validFrom).max
timestamp : Instant = 2023-06-22T12:42:10.034684Z
```

Take the `max` of the two timestamps which corresponds to the `RequestSide.From` topology transaction that you added above. Use this timestamp now to export the state using:

```
@ participant1.repair.download(Set(alice), "alice.acs.gz", filterDomainId=
↪"mydomain", timestamp = Some(timestamp))
```

Note that you need to do this for every domain separately with the correct timestamp of the activation of the party. In our example, there is only one domain.

Subsequently, the active contract set is imported on the target participant:

```
@ repair.party_migration.step2_import_acs(participant2, "alice.acs.gz")
```

Once the entire active contract store has been imported, the target participant can reconnect to the domain:

```
@ participant2.domains.reconnect_all()
```

Now, both participant host the party and can act on behalf of it.

1.30.3.5 Manually Initializing a Node

There are situations where a node should not be automatically initialized, but where we prefer to control each step of the initialization. For example, when a node in the setup does not control its own identity, or when we do not want to store the identity key on the node for security reasons.

In the following, we demonstrate the basic steps how to initialise a node:

Keys Initialization

The following steps describe how to manually generate the necessary Canton keys (e.g. for a participant):

```
// first, let's create a signing key that is going to control our identity.
val identityKey =
  participant.keys.secret.generate_signing_key(name = participant.name + "-
↪namespace")

// create signing and encryption keys
val signingKey =
  participant.keys.secret.generate_signing_key(name = participant.name + "-signing
↪")
val encryptionKey =
  participant.keys.secret.generate_encryption_key(name = participant.name + "-
↪encryption")
```

Domain Initialization

The following steps describe how to manually initialize a domain node:

```
// use the fingerprint of this key for our identity
val namespace = identityKey.fingerprint

// initialise the identity of this domain
val uid = mydomain.topology.init_id(identifier = "mydomain", fingerprint =
↪namespace)

// create the root certificate for this namespace
mydomain.topology.namespace_delegations.authorize(
  ops = TopologyChangeOp.Add,
  namespace = namespace,
  authorizedKey = namespace,
  isRootDelegation = true,
)

// set the initial dynamic domain parameters for the domain
mydomain.topology.domain_parameters_changes
  .authorize(
    domainId = DomainId(uid),
    newParameters =
      ConsoleDynamicDomainParameters.defaultValues(protocolVersion =
↪testedProtocolVersion),
    protocolVersion = testedProtocolVersion,
```

(continues on next page)

(continued from previous page)

```
  )

val mediatorId = MediatorId(uid)
Seq[Member](DomainTopologyManagerId(uid), SequencerId(uid), mediatorId).foreach {
↪keyOwner =>
  // in this case, we are using an embedded domain. therefore, we initialise all
↪domain
  // entities at once. in a distributed setup, the process needs to be invoked on
  // the separate entities, and therefore requires a bit more coordination.
  // however, the steps remain the same.

  // then, create a topology transaction linking the entity to the signing key
  mydomain.topology.owner_to_key_mappings.authorize(
    ops = TopologyChangeOp.Add,
    keyOwner = keyOwner,
    key = signingKey.fingerprint,
    purpose = KeyPurpose.Signing,
  )
}

// Register the mediator
mydomain.topology.mediator_domain_states.authorize(
  ops = TopologyChangeOp.Add,
  domain = mydomain.id,
  mediator = mediatorId,
  side = RequestSide.Both,
)
```

Participant Initialization

The following steps describe how to manually initialize a participant node:

```
// use the fingerprint of this key for our identity
val namespace = identityKey.fingerprint

// create the root certificate (self-signed)
participant.topology.namespace_delegations.authorize(
  ops = TopologyChangeOp.Add,
  namespace = namespace,
  authorizedKey = namespace,
  isRootDelegation = true,
)

// initialise the id: this needs to happen AFTER we created the namespace
↪delegation
// (on participants; for the domain, it's the other way around ... sorry for that)
// if we initialize the identity before we added the root certificate, then the
↪system will
// complain about not being able to vet the admin workflow packages automatically.
// that would not be tragic, but would require a manual vetting step.
// in production, use a "random" identifier. for testing and development, use
↪something
// helpful so you don't have to grep for hashes in your log files.
participant.topology.init_id(
```

(continues on next page)

(continued from previous page)

```
  identifier = Identifier.tryCreate("manualInit"),
  fingerprint = namespace,
)

// assign new keys to this participant
Seq(encryptionKey, signingKey).foreach { key =>
  participant.topology.owner_to_key_mappings.authorize(
    ops = TopologyChangeOp.Add,
    keyOwner = participant.id,
    key = key.fingerprint,
    purpose = key.purpose,
  )
}
```

1.31 Common Operational Tasks

1.31.1 Manage Dars and Packages

A package is a unit of compiled Daml code corresponding to one Daml project. A DAR is a collection of packages including a main package and all other packages from the dependencies of this Daml project.

1.31.1.1 Uploading DARs

To use a Daml application on a participant, you need to upload it to your participant node. The application always comes packaged as one or more DARs that need to be uploaded in the order of their dependency. There are two ways to upload DARs to a Canton node: either via the Ledger API, or through Canton *console command*:

```
@ participant2.dars.upload("dars/CantonExamples.dar")
res1: String =
↪"1220c783022e36adf132a905711d40850477d4b817e39f1b44d62af0f4a7a3c05476"
```

1.31.1.2 Inspecting DARs and Packages

You can get a list of uploaded DARs using:

```
@ participant2.dars.list()
res2: Seq[com.digitalasset.canton.participant.admin.v0.DarDescription] = Vector(
  DarDescription(
    hash = "1220c783022e36adf132a905711d40850477d4b817e39f1b44d62af0f4a7a3c05476",
    name = "CantonExamples"
  ),
  DarDescription(
    hash = "122012a6f2b7c0b666e7541ce6f5d4273ab8d00da671b4d3bbb9bebb6a5120ec02c5",
    name = "AdminWorkflowsWithVacuuming"
  )
)
```

Please note that the package AdminWorkflows is a package that ships with Canton. It contains the Daml templates used by the `participant.health.ping` command.

In order to inspect the contents of the DAR, you need to grab the hash identifying it:

```
@ val dars = participant2.dars.list(filterName = "CantonExamples")
dars : Seq[com.digitalasset.canton.participant.admin.v0.DarDescription] = Vector(
  DarDescription(
    hash = "1220c783022e36adf132a905711d40850477d4b817e39f1b44d62af0f4a7a3c05476",
    name = "CantonExamples"
  )
)
```

```
@ val hash = dars.head.hash
hash : String =
↪"1220c783022e36adf132a905711d40850477d4b817e39f1b44d62af0f4a7a3c05476"
```

Using that hash, you can inspect the contents of the DAR using:

```
@ val darContent = participant2.dars.list_contents(hash)
darContent : DarMetadata = DarMetadata(
  name = "CantonExamples",
  main = "9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0",
  packages = Vector(
    "9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0",
    "bef3d1e9c2f8be31f80c032e930c85e336da27b64ebb1e3a31c9072e9df3a14b",
    "cb0552debf219cc909f51cbb5c3b41e9981d39f8f645b1f35e2ef5be2e0b858a",
    "3f4deaf145a15cdcfa762c058005e2edb9baa75bb7f95a4f8f6f937378e86415",
..
```

You can also directly look at the packages, using:

```
@ participant2.packages.list()
res6: Seq[com.digitalasset.canton.participant.admin.v0.PackageDescription] =
↪Vector(
  PackageDescription(
    packageId = "86828b9843465f419db1ef8a8ee741d1eef645df02375ebf509cdc8c3ddd16cb
↪",
    sourceDescription = "CantonExamples"
  ),
  PackageDescription(
    packageId = "cc348d369011362a5190fe96dd1f0dfbc697fdfd10e382b9e9666f0da05961b7
↪",
    sourceDescription = "CantonExamples"
..
```

Please note that a DAR can include packages that are already included in other DARs. In particular the Daml standard library are shipped with every DAR. Therefore, the `sourceDescription` will always contain only one textual reference to a DAR.

You can also inspect the content of a package, using:

```
@ participant2.packages.list_contents(darContent.main)
res7: Seq[com.digitalasset.canton.participant.admin.v0.ModuleDescription] =
↪Vector(
  ModuleDescription(name = "CantonExamples"),
  ModuleDescription(name = "ContractKeys"),
```

(continues on next page)

(continued from previous page)

```
  ModuleDescription(name = "SafePaint"),
  ModuleDescription(name = "LockIou"),
  ModuleDescription(name = "Iou"),
  ModuleDescription(name = "Divulgence"),
  ModuleDescription(name = "Paint"),
..
```

1.31.1.3 Understanding Package Vetting

Every participant operator uploads DARs individually to their participant node. There is no global DAR repository anywhere and participants do not have access to each others DAR repositories. Therefore, for two participants to synchronise on a transaction that uses packages contained in a certain DAR, we need both participant operators to have uploaded the same DAR before the transaction was submitted.

If one of the involved participants doesn't know about a certain DAR, then the transaction will bounce with an error PACKAGE_NOT_VETTED_BY_RECIPIENTS.

This error goes back to the fact that both participants not only upload the DAR, but also publicly declare towards their peers that they are ready to receive transactions referring to certain packages. This declaration happens automatically when you upload a DAR. The package vettings can be inspected using (preview):

```
@ participant2.topology.vetted_packages.list()
res8: Seq[ListVettedPackagesResult] = Vector(
  ListVettedPackagesResult(
    context = BaseResult(
      domain = "Authorized",
      validFrom = 2023-06-12T12:18:42.142697Z,
      validUntil = None,
      operation = Add,
      serialized = <ByteString@110816a5 size=2582 contents="\n\223\024\n\301\021\
↪n\274\021\n\271\021\022 QJVnZH6yMsV48KljIgN1QyQ53uSqnNBtJ...">,
..
```

Vetting is necessary, as otherwise, a malicious participant might send a transaction referring to package a receiver does not have, which would make it impossible for the receiver to process the transaction, leading to a ledger fork. As transactions are valid only if all involved participants have vetted the used packages, this attack cannot happen.

1.31.1.4 Removing Packages and DARs

Note: Note that package and DAR removal is under active development. The behaviour described in this documentation may change in the future. Package and DAR removal is a preview feature and should not be used in production.

Canton supports removal of both packages and DARs that are no longer in use. Removing unused packages and DARs has the following advantages:

Freeing up storage

Preventing accidental use of the old package / DAR
Reducing the number of packages / DARs that are trusted and may potentially have to be audited

Certain conditions must to be met in order to remove packages or DARs. These conditions are designed to prevent removal of packages or DARs that are currently in use. The rest of this page describes the requirements.

Removing DARs

The following checks are performed before a DAR can be removed:

The main package of the DAR must be unused – there should be no active contract from this package
All package dependencies of the DAR should either be unused or contained in another of the participant node's uploaded DARs. Canton uses this restriction to ensure that the package dependencies of the DAR don't become stranded if they're in use.
The main package of the dar should not be vetted. If it is vetted, Canton will try to automatically revoke the vetting for the main package of the DAR, but this automatic vetting revocation will only succeed if the main package vetting originates from a standard `dars.upload`. Even if the automatic revocation fails, you can always manually revoke the package vetting.

The following tutorial shows how to remove a DAR with the Canton console. The first step is to upload a DAR so that we have one to remove. Additionally, store the packages that are present before the DAR is uploaded, as these can be used to double-check that DAR removal reverts to a clean state.

```
@ val packagesBefore = participant1.packages.list().map(_.packageId).toSet
packagesBefore : Set[String] = HashSet(
  "86828b9843465f419db1ef8a8ee741d1eef645df02375ebf509cdc8c3ddd16cb",
  "5921708ce82f4255deb1b26d2c05358b548720938a5a325718dc69f381ba47ff",
  "cc348d369011362a5190fe96dd1f0dfbc697fdfd10e382b9e9666f0da05961b7",
  "bef3d1e9c2f8be31f80c032e930c85e336da27b64ebb1e3a31c9072e9df3a14b",
  "6839a6d3d430c569b2425e9391717b44ca324b88ba621d597778811b2d05031d",
  "99a2705ed38c1c26cbb8fe7acf36bbf626668e167a33335de932599219e0a235",
  "e22bce619ae24ca3b8e6519281cb5a33b64b3190cc763248b4c3f9ad5087a92c",
  "d58cf9939847921b2aab78eaa7b427dc4c649d25e6bee3c749ace4c3f52f5c97",
  "6c2c0667393c5f92f1885163068cd31800d2264eb088eb6fc740e11241b2bf06",
..
```

```
@ val darHash = participant1.dars.upload("dars/CantonExamples.dar")
darHash : String =
 ↪"1220c783022e36adf132a905711d40850477d4b817e39f1b44d62af0f4a7a3c05476"
```

If the DAR hash is unknown, it can be found using `dars.list`:

```
@ val darHash_ = participant1.dars.list().filter(_.name == "CantonExamples").head.
 ↪hash
darHash_ : String =
 ↪"1220c783022e36adf132a905711d40850477d4b817e39f1b44d62af0f4a7a3c05476"
```

The DAR can then be removed with the following command:

```
@ participant1.dars.remove(darHash)
```

Note that, right now, DAR removal will only remove the main packages associated with the DAR:

```
@ val packageIds = participant1.packages.list().filter(_.sourceDescription ==
↪"CantonExamples").map(_.packageId)
packageIds : Seq[String] = Vector(
  "86828b9843465f419db1ef8a8ee741d1eef645df02375ebf509cdc8c3ddd16cb",
  "cc348d369011362a5190fe96dd1f0dfbc697fdfd10e382b9e9666f0da05961b7",
  "e491352788e56ca4603acc411ffe1a49fefd76ed8b163af86cf5ee5f4c38645b",
  "cb0552debf219cc909f51cbb5c3b41e9981d39f8f645b1f35e2ef5be2e0b858a",
  "38e6274601b21d7202bb995bc5ec147decda5a01b68d57dda422425038772af7",
  "99a2705ed38c1c26cbb8fe7acf36bbf626668e167a33335de932599219e0a235",
  "f20de1e4e37b92280264c08bf15eca0be0bc5babd7a7b5e574997f154c00cb78",
  "283fdcf3bbbc04db4ee15ba5760dbe459aee1087f358b7e6cd4d7da2ff36e776",
  "8a7806365bbd98d88b4c13832ebfa305f6abaeaf32cfa2b7dd25c4fa489b79fb",
..
```

It's possible to remove each of these manually, using package removal. There is a complication here that packages needed for admin workflows (e.g. the Ping command) cannot be removed, so these are skipped.

```
@ packageIds.filter(id => ! packagesBefore.contains(id)).foreach(id =>
↪participant1.packages.remove(id))
```

The following command verifies that all the packages have been removed.

```
@ val packages = participant1.packages.list().map(_.packageId).toSet
packages : Set[String] = HashSet(
  "86828b9843465f419db1ef8a8ee741d1eef645df02375ebf509cdc8c3ddd16cb",
  "5921708ce82f4255deb1b26d2c05358b548720938a5a325718dc69f381ba47ff",
  "cc348d369011362a5190fe96dd1f0dfbc697fdfd10e382b9e9666f0da05961b7",
  "bef3d1e9c2f8be31f80c032e930c85e336da27b64ebb1e3a31c9072e9df3a14b",
  "6839a6d3d430c569b2425e9391717b44ca324b88ba621d597778811b2d05031d",
  "99a2705ed38c1c26cbb8fe7acf36bbf626668e167a33335de932599219e0a235",
  "e22bce619ae24ca3b8e6519281cb5a33b64b3190cc763248b4c3f9ad5087a92c",
  "d58cf9939847921b2aab78eaa7b427dc4c649d25e6bee3c749ace4c3f52f5c97",
  "6c2c0667393c5f92f1885163068cd31800d2264eb088eb6fc740e11241b2bf06",
..
```

```
@ assert(packages == packagesBefore)
```

The following sections explain what happens when the DAR removal operation goes wrong, for various reasons.

Main package of the DAR is in use

The first step to illustrate this is to upload a DAR and create a contract using the main package of the DAR:

```
@ val darHash = participant1.dars.upload("dars/CantonExamples.dar")
darHash : String =
↪"1220c783022e36adf132a905711d40850477d4b817e39f1b44d62af0f4a7a3c05476"
```

```
@ val packageId = participant1.packages.find("Iou").head.packageId
packageId : String =
↪"9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0"
```

```
@ participant1.domains.connect_local(mydomain)
```

```
@ val createIouCmd = ledger_api_utils.create(packageId,"Iou","Iou",Map("payer" ->
↪participant1.adminParty,"owner" -> participant1.adminParty,"amount" -> Map(
↪"value" -> 100.0, "currency" -> "EUR"),"viewers" -> List()))
..
```

```
@ participant1.ledger_api.commands.submit(Seq(participant1.adminParty),
↪Seq(createIouCmd))
res21: com.daml.ledger.api.v1.transaction.TransactionTree = TransactionTree(
  transactionId =
↪"1220140615c40a381f9b867ceb78961bb1fbaceb82c8c52259ce4c5e83940bd4fc4e",
  commandId = "09fd6428-b7a8-49eb-9972-85f1f3dd9376",
  workflowId = "",
  effectiveAt = Some(
..
```

Now that a contract exists using the main package of the DAR, a subsequent DAR removal operation will fail:

```
@ participant1.dars.remove(darHash)
ERROR com.digitalasset.canton.integration.EnterpriseEnvironmentDefinition$$anon$3
↪- Request failed for participant1.
  GrpcRequestRefusedByServer: FAILED_PRECONDITION/PACKAGE_OR_DAR_REMOVAL_ERROR(9,
↪40ff158c): The DAR DarDescriptor(SHA-256:c783022e36ad...,CantonExamples) cannot
↪be removed because its main package
↪9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0 is in-use by
↪contract
↪ContractId(005170f294b69a37a7ba0c30a8f0c6ea1ab81e142e74fb146f19104af801cac302ca0112203845
on domain mydomain::1220bf7c580f....
  Request:
↪RemoveDar(1220c783022e36adf132a905711d40850477d4b817e39f1b44d62af0f4a7a3c05476)
  CorrelationId: 40ff158cd233c870d3dcba1e95b267bb
  Context: HashMap(participant -> participant1, test ->
↪PackageDarManagementDocumentationIntegrationTest, pkg ->
↪9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0, tid ->
↪40ff158cd233c870d3dcba1e95b267bb)
  Command ParticipantAdministration$dars$.remove invoked from cmd10000056.sc:1
```

In order to remove the DAR, we must archive this contract. Note that the contract ID for this contract can also be found in the error message above.

```
@ val iou = participant1.ledger_api.acs.find_generic(participant1.adminParty, _.
↪templateId.isModuleEntity("Iou", "Iou"))
iou : com.digitalasset.canton.admin.api.client.commands.LedgerApiTypeWrappers.
↪WrappedCreatedEvent = WrappedCreatedEvent(
  event = CreatedEvent(
    eventId = "
↪#1220140615c40a381f9b867ceb78961bb1fbaceb82c8c52259ce4c5e83940bd4fc4e:0",
    contractId =
↪"005170f294b69a37a7ba0c30a8f0c6ea1ab81e142e74fb146f19104af801cac302ca0112203845e89891f897
↪",
    templateId = Some(
      value = Identifier(
        packageId =
↪"9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0",
```

(continues on next page)

(continued from previous page)

```
        moduleName = "Iou",
        entityName = "Iou"
      )
..
```

```
@ val archiveIouCmd = ledger_api_utils.exercise("Archive", Map.empty, iou.event)
..
```

```
@ participant1.ledger_api.commands.submit(Seq(participant1.adminParty),
↪Seq(archiveIouCmd))
res24: com.daml.ledger.api.v1.transaction.TransactionTree = TransactionTree(
  transactionId =
↪"1220e94572b389df0216fefdbbd67933938779a64c789fa2a2fd01a9ad19ea34125d",
  commandId = "a638b802-02b4-4fdb-a4f6-a3d59a6777f5",
  workflowId = "",
  effectiveAt = Some(
..
```

The DAR removal operation will now succeed.

```
@ participant1.dars.remove(darHash)
```

Main package of the DAR can't be automatically removed

Similarly, DAR removal may fail because the DAR can't be automatically removed. To illustrate this, upload the DAR without automatic vetting and subsequently vet all the packages manually.

```
@ val darHash = participant1.dars.upload("dars/CantonExamples.dar",
↪vetAllPackages = false)
darHash : String =
↪"1220c783022e36adf132a905711d40850477d4b817e39f1b44d62af0f4a7a3c05476"
```

```
@ import com.daml.lf.data.Ref.IdString.PackageId
```

```
@ val packageIds = participant1.packages.list().filter(_.sourceDescription ==
↪"CantonExamples").map(_.packageId).map(PackageId.assertFromString)
packageIds : Seq[PackageId] = Vector(
  "86828b9843465f419db1ef8a8ee741d1eef645df02375ebf509cdc8c3ddd16cb",
  "cc348d369011362a5190fe96dd1f0dfbc697fdfd10e382b9e9666f0da05961b7",
..
```

```
@ participant1.topology.vetted_packages.authorize(TopologyChangeOp.Add,
↪participant1.id, packageIds)
res29: com.google.protobuf.ByteString = <ByteString@27db41fd size=2382 contents="\
↪n\313\022\n\373\017\n\366\017\n\363\017\022 0SDxlEAROOtxd441yH4iwrMCwtqn7ZB2J...
↪">
```

The DAR removal operation will now fail:

```
@ participant1.dars.remove(darHash)
ERROR com.digitalasset.canton.integration.EnterpriseEnvironmentDefinition$$anon$3
↪- Request failed for participant1.
```

(continues on next page)

(continued from previous page)

```
  GrpcRequestRefusedByServer: FAILED_PRECONDITION/PACKAGE_OR_DAR_REMOVAL_ERROR(9,
↪6d26d83c): An error was encountered whilst trying to unvet the DAR
↪DarDescriptor(SHA-256:c783022e36ad...,CantonExamples) with main package
↪9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0 for DAR
↪removal. Details: IdentityManagerParentError(Mapping(VettedPackages(
  participant = participant1::12203c8338b7...,
  packages = Seq(
    9d65f326a67a...,
    bef3d1e9c2f8...,
    cb0552debf21...,
    3f4deaf145a1...,
    86828b984346...,
    f20de1e4e37b...,
    76bf0fd12bd9...,
    38e6274601b2...,
    d58cf...
  Request:
↪RemoveDar(1220c783022e36adf132a905711d40850477d4b817e39f1b44d62af0f4a7a3c05476)
  CorrelationId: 6d26d83cdf69a4e4f41f56f2d3f1e28c
  Context: Map(participant -> participant1, tid ->
↪6d26d83cdf69a4e4f41f56f2d3f1e28c, test ->
↪PackageDarManagementDocumentationIntegrationTest)
  Command ParticipantAdministration$dars$.remove invoked from cmd10000076.sc:1
```

The DAR can be successfully removed after manually revoking the vetting for the main package:

```
@ participant1.topology.vetted_packages.authorize(TopologyChangeOp.Remove,
↪participant1.id, packageIds, force = true)
res30: com.google.protobuf.ByteString = <ByteString@237238b1 size=2384 contents="\
↪n\315\022\n\375\017\n\370\017\n\365\017\b\001\022
↪0SDxlEAROOtxd441yH4iwrMCwtqn7ZB...">
```

```
@ participant1.dars.remove(darHash)
```

Note that a `force` flag is needed used to revoke the package vetting; throughout this tutorial `force` will be used whenever a package vetting is being removed. See *topology.vetted_packages.authorize* for more detail.

Removing Packages

Canton also supports removing individual packages, giving the user more fine-grained control over the system. Packages can be removed if the package satisfies the following two requirements:

The package must be unused. This means that there shouldn't be an active contract corresponding to the package.
The package must not be vetted. This means there shouldn't be an active vetting transaction corresponding to the package.

The following tutorial shows how to remove a package using the Canton console. The first step is to upload and identify the package ID for the package to be removed.

```
@ val darHash = participant1.dars.upload("dars/CantonExamples.dar")
darHash : String =
↪"1220c783022e36adf132a905711d40850477d4b817e39f1b44d62af0f4a7a3c05476"
```

```
@ val packageId = participant1.packages.find("Iou").head.packageId
packageId : String =
↪"9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0"
```

Package removal will initially fail as, by default, uploading the DAR will add a vetting transaction for the package:

```
@ participant1.packages.remove(packageId)
ERROR com.digitalasset.canton.integration.EnterpriseEnvironmentDefinition$$anon$3
↪- Request failed for participant1.
  GrpcRequestRefusedByServer: FAILED_PRECONDITION/PACKAGE_OR_DAR_REMOVAL_ERROR(9,
↪a10cdd12): Package
↪9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0 is currently
↪vetted and available to use.
  Request:
↪RemovePackage(9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0,
↪false)
  CorrelationId: a10cdd12ceeefbc4a17c2bc21b469371
  Context: Map(participant -> participant1, tid ->
↪a10cdd12ceeefbc4a17c2bc21b469371, test ->
↪PackageDarManagementDocumentationIntegrationTest)
  Command ParticipantAdministration$packages$.remove invoked from cmd10000087.sc:1
```

The vetting transaction must be manually revoked:

```
@ val packageIds = participant1.topology.vetted_packages.list().map(_.item.
↪packageIds).filter(_.contains(packageId)).head
packageIds : Seq[com.digitalasset.canton.package.LfPackageId] = Vector(
  "9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0",
  "bef3d1e9c2f8be31f80c032e930c85e336da27b64ebb1e3a31c9072e9df3a14b",
..
```

```
@ participant1.topology.vetted_packages.authorize(TopologyChangeOp.Remove,
↪participant1.id, packageIds, force = true)
res35: com.google.protobuf.ByteString = <ByteString@4fb290fb size=2384 contents="\
↪n\315\022\n\375\017\n\370\017\n\365\017\b\001\022
↪mpi2lnnmh4OW4bQLCDbKw64QaIc9acP...">
```

And then the package can be removed:

```
@ participant1.packages.remove(packageId)
```

Package is in use

The operations above will fail if the package is in use. To illustrate this, first re-upload the package (uploading the associated DAR will work):

```
@ val darHash = participant1.dars.upload("dars/CantonExamples.dar")
darHash : String =
↪"1220c783022e36adf132a905711d40850477d4b817e39f1b44d62af0f4a7a3c05476"
```

Then create a contract using the package:

```
@ val createIouCmd = ledger_api_utils.create(packageId,"Iou","Iou",Map("payer" ->
↪participant1.adminParty,"owner" -> participant1.adminParty,"amount" -> Map(
↪"value" -> 100.0, "currency" -> "EUR"),"viewers" -> List()))
createIouCmd : com.daml.ledger.api.v1.commands.Command = Command(
  command = Create(
    value = CreateCommand(
      templateId = Some(
        value = Identifier(
..
```

```
@ participant1.ledger_api.commands.submit(Seq(participant1.adminParty),
↪Seq(createIouCmd))
res39: com.daml.ledger.api.v1.transaction.TransactionTree = TransactionTree(
  transactionId =
↪"12204b37add31c559ac455ef8b81c5b2b4b4a9fc582aa8af26312b9fdcff5f8f0722",
  commandId = "9c107e7c-e63a-46af-bfbc-7dc36d0e6a31",
  workflowId = "",
  effectiveAt = Some(
    value = Timestamp(
      seconds = 1686572343L,
      nanos = 324270000,
      unknownFields = UnknownFieldSet(fields = Map())
    )
..
```

In this situation, the package cannot be removed:

```
@ participant1.packages.remove(packageId)
ERROR com.digitalasset.canton.integration.EnterpriseEnvironmentDefinition$$anon$3
↪- Request failed for participant1.
  GrpcRequestRefusedByServer: FAILED_PRECONDITION/PACKAGE_OR_DAR_REMOVAL_ERROR(9,
↪7f16ca92): Package
↪9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0 is currently
↪in-use by contract
↪ContractId(00f83a2f1d08029ba57a7d094db04e384e65ca77db569936e4b4ead52805993eb4ca0112209b30
↪on domain mydomain::1220bf7c580f.... It may also be in-use by other contracts.
  Request:
↪RemovePackage(9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0,
↪false)
  CorrelationId: 7f16ca92dea13a8d2e27e2a68d3ed5fe
  Context: HashMap(participant -> participant1, test ->
↪PackageDarManagementDocumentationIntegrationTest, domain ->
↪mydomain::1220bf7c580f..., pkg ->
↪9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0, tid ->
↪7f16ca92dea13a8d2e27e2a68d3ed5fe, contract ->
↪ContractId(00f83a2f1d08029ba57a7d094db04e384e65ca77db569936e4b4ead52805993eb4ca0112209b30
  Command ParticipantAdministration$packages$.remove invoked from cmd10000103.sc:1
```

To remove the package, first archive the contract:

```
@ val iou = participant1.ledger_api.acs.find_generic(participant1.adminParty, _.
↪templateId.isModuleEntity("Iou", "Iou"))
iou : com.digitalasset.canton.admin.api.client.commands.LedgerApiTypeWrappers.
↪WrappedCreatedEvent = WrappedCreatedEvent(
  event = CreatedEvent(
    eventId = "
↪#12204b37add31c559ac455ef8b81c5b2b4b4a9fc582aa8af26312b9fdcff5f8f0722:0"
```

(continues on next page)

(continued from previous page)

```
    contractId =
↪"00f83a2f1d08029ba57a7d094db04e384e65ca77db569936e4b4ead52805993eb4ca0112209b309891c94663
↪",
    templateId = Some(
      value = Identifier(
        packageId =
↪"9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0",
        moduleName = "Iou",
        entityName = "Iou"
      )
..
```

```
@ val archiveIouCmd = ledger_api_utils.exercise("Archive", Map.empty, iou.event)
archiveIouCmd : com.daml.ledger.api.v1.commands.Command = Command(
  command = Exercise(
    value = ExerciseCommand(
      templateId = Some(
        value = Identifier(
          packageId =
↪"9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0",
          moduleName = "Iou",
          entityName = "Iou"
        )
      ),
..
```

```
@ participant1.ledger_api.commands.submit(Seq(participant1.adminParty),
↪Seq(archiveIouCmd))
res42: com.daml.ledger.api.v1.transaction.TransactionTree = TransactionTree(
  transactionId =
↪"1220fb7be1e400fe1f2d4506b8c5fa1c383e200150667e1feffbbe6b81f4b23695a1",
  commandId = "9605fb99-b2f5-4483-aadf-82f9464041e3",
  workflowId = "",
  effectiveAt = Some(
    value = Timestamp(
      seconds = 1686572344L,
      nanos = 187726000,
      unknownFields = UnknownFieldSet(fields = Map())
    )
  ),
  offset = "00000000000000000c",
..
```

Then revoke the package vetting transaction:

```
@ val packageIds = participant1.topology.vetted_packages.list().map(_.item.
↪packageIds).filter(_.contains(packageId)).head
packageIds : Seq[com.digitalasset.canton.package.LfPackageId] = Vector(
  "9d65f326a67a0dc9a723dbaa3abb1b67831858940cfe6376475d7959120fe6d0",
  "bef3d1e9c2f8be31f80c032e930c85e336da27b64ebb1e3a31c9072e9df3a14b",
..
```

```
@ participant1.topology.vetted_packages.authorize(TopologyChangeOp.Remove,
↪participant1.id, packageIds, force = true)
```

(continues on next page)

(continued from previous page)

```
res44: com.google.protobuf.ByteString = <ByteString@6328d67d size=2384 contents="\
↪n\315\022\n\375\017\n\370\017\n\365\017\b\001\022
↪LJkyooFPhwkMj4HzoHpmsrxdXvEDWsP..."">
```

The package removal operation should now succeed.

```
@ participant1.packages.remove(packageId)
```

Force-removing packages

Packages can also be forcibly removed, even if the conditions above are not satisfied. This is done by setting the `force` flag to `true`.

To experiment with this, first re-upload the DAR so the package becomes available again:

```
@ participant1.dars.upload("dars/CantonExamples.dar")
res46: String =
↪"1220c783022e36adf132a905711d40850477d4b817e39f1b44d62af0f4a7a3c05476"
```

Then force-remove the package:

```
@ participant1.packages.remove(packageId, force = true)
```

Please note, this is a dangerous operation. Forced removal of packages should be avoided whenever possible.

1.31.2 Upgrading

This section covers the processes to upgrade Canton participant and domain nodes. Upgrading Daml applications is covered elsewhere.

As elaborated in the *versioning guide*, new features, improvements and fixes are released regularly. To benefit from these changes, the Canton-based system must be upgraded.

There are two key aspects that need to be addressed when upgrading a system:

Upgrading the Canton binary that is used to run a node.
Upgrading the protocol version (wire format and semantics of the APIs used between the nodes).

Canton is a distributed system, where no single operator controls all nodes. Therefore, we must support the situation where nodes are upgraded individually, providing a safe upgrade mechanism that requires the minimal amount of synchronized actions within a network.

A Canton binary supports *multiple protocol versions*, and new protocol versions are introduced in a backwards compatible way with a new binary (see *version table*). Therefore, any upgrade of a protocol used in a distributed Canton network is done by individually upgrading all binaries and subsequently changing the protocol version used among the nodes to the desired one.

The following recipe is a general guide. Before upgrading to a specific version, please check the individual notes for each version.

This guide also assumes that the upgrade is a minor or a patch release. Major release upgrades might differ and will be covered separately if necessary.

Please read the entire guide before proceeding, please backup your data before you do any upgrade, and please test your upgrade carefully before attempting to upgrade your production system.

1.31.2.1 Upgrade Canton Binary

A Canton node consists of one or more processes, where each process is defined by

A Java Virtual Machine application running a versioned jar of Canton.
A set of configuration files describing the node that is being run.
An optional bootstrap script passed via `--boostrap`, which runs on startup.
A database (with a specific schema), holding the data of the node.

Therefore, to upgrade the node, you will need to not only replace the jar, but also test that the configuration files can still be parsed by the new process, that the bootstrap script you are using is still working, and you need to upgrade the database schema.

Generally, all changes to configuration files should be backwards compatible, and therefore not be affected by the upgrade process. In rare cases, there might be a minor change to the configuration file necessary in order to support the upgrade process. Sometimes, fixing a substantial bug might require a minor breaking change to the API. The same applies to Canton scripts.

The schema in the database is versioned and managed using Flyway. Detecting and applying changes is done by Canton using that library. Understanding this background can be helpful to troubleshoot issues.

Preparation

First, please download the new Canton binary that you want to upgrade to and store it on the test system where you want to test the upgrade process first.

Then, obtain a recent backup of the database of the node and deploy it to a database server of your convenience, such that **you can test the upgrade process without affecting your production system**. While we extensively test the upgrade process ourselves, we cannot exclude the eventuality that you are using the system in a non-anticipated way. Testing is cumbersome, but breaking a production system is worse.

If you are upgrading a participant, then we suggest that you also use an in-memory domain which you can tear down after you've tested that the upgrade of the participant is working. You might do that by adding a simple domain definition as a configuration mixin to your participant configuration.

Generally, if you are running an high-availability setup, please take all nodes offline before performing an upgrade. If the update requires a database migration (check the release notes), avoid running older and newer binaries in a replicated setup, as the two binaries might expect a different database layout.

You can upgrade the binaries of a microservice-based domain in any order, as long as you upgrade the binaries of nodes accessing the same database at the same time. For example, you could upgrade the binary of a replicated mediator node on one weekend and an active-active database sequencer on another weekend.

Back Up Your Database

Before you upgrade the database and binary, please ensure that you have backed up your data, such that you can roll back to the previous version in case of an issue. You can backup your data by cloning it. In Postgres, the command is:

```
CREATE DATABASE newdb WITH TEMPLATE originaldb OWNER dbuser;
```

When doing this, you need to change the database name and user name in above command to match your setup.

Test your Configuration

First, let's test that the configuration still works

```
./bin/canton -v -c storage-for-upgrade-testing.conf -c mynode.conf --manual-start
```

Here, the files `storage-for-upgrade-testing.conf` and `mynode.conf` need to be adjusted to match your case.

If Canton starts and shows the command prompt of the console, then the configuration was parsed successfully.

The command line option `--manual-start` will ensure that the node is not started automatically, as we first need to migrate the database.

Migrating the Database

Canton does not perform a database migration automatically. Migrations need to be forced. If you start a node with that requires a database migration, you will observe the following Flyway error:

```
@ participant.start()
ERROR com.digitalasset.canton.integration.EnterpriseEnvironmentDefinition$$anon$3
 ↪- failed to initialize participant: There are 5 pending migrations to get to
 ↪database schema version 6. Currently on version 1.1. Please run `participant.db.
 ↪migrate` to apply pending migrations
  Command LocalParticipantReference.start invoked from cmd10000002.sc:1
```

The database schema definitions are versioned and hashed. This error informs us about the current database schema version and how many migrations need to be applied.

We can now force the migration to a new schema using:

```
@ participant.db.migrate()
```

Please note that you need to ensure that the user account the node is using to access the database allows to change the database schema. How long the migration takes depends on the version of the binary (see migration notes), the size of the database and the performance of the database server.

Subsequently, you can successfully start the node

```
@ participant.start()
```

Please note that while we've used a participant node here as an example, the behaviour is the same for all other types of nodes.

Test Your Upgrade

Once your node is up and running, you can test it by running a ping. If you are testing the upgrade of your participant node, then you might want to connect to the test domain

```
@ testdomain.start()
```

```
@ participant.domains.connect_local(testdomain)
```

If you did the actual upgrade of the production instance, then you would just reconnect to the current domain before running the ping:

```
@ participant.domains.reconnect_all()
```

You can check that the domain is up and running using

```
@ participant.domains.list_connected()
res6: Seq[ListConnectedDomainsResult] = Vector(
  ListConnectedDomainsResult(
    domainAlias = Domain 'testdomain',
    domainId = testdomain::122056e307f0...,
    healthy = true
  )
)
```

Finally, you can ping the participant to see if the system is operational

```
@ participant.health.ping(participant)
res7: Duration = 987 milliseconds
```

Version Specific Notes

Upgrade to Release 2.7

Version 2.7 slightly extends the database schema. Therefore, you will have to perform the *database migration steps*. Alternatively, you can enable the new migrate and start mode in Canton, which triggers an automatic update of the database schema when a new minor version is deployed. This mode can be enabled by setting the appropriate storage parameter:

```
canton.X.Y.storage.parameters.migrate-and-start = yes
```

To benefit from the new security features in protocol version 5, you must *upgrade the domain accordingly*.

Activation of unsupported features

In order to activate unsupported features, you now need to explicitly enable *dev-version-support* on the domain (in addition to the non-standard config flag). More information can be found in the *documentation*.

Breaking changes around console commands

Key rotation The command `keys.secret.rotate_wrapper_key` now returns a different error code. An `INVALID_WRAPPER_KEY_ID` error has been replaced by an `INVALID_KMS_KEY_ID` error.

Adding sequencer connection The configuration of the sequencer client has been updated to accommodate multiple sequencers and their endpoints: method `addConnection` has been renamed to `addEndpoints` to better reflect the fact that it modifies an endpoint for the sequencer.

Hence, command to add a new sequencer connection to the mediator would be changed to:

```
mediator1.sequencer_connection.modifyConnections(
    _.addEndpoints(SequencerAlias.Default, connection)
)
```

Unique contract key deprecation

The unique-contract-keys parameters for both participant and sync domain nodes are now marked as deprecated. As of this release, the meaning and default value (true) remain unchanged. However, contract key uniqueness will not be available in the next major version, featuring multi-domain connectivity. If you are already setting this key to false explicitly (preview), this behavior will be the default one after the configuration key is removed. If you don't explicitly set this value to false, you are encouraged to evaluate evolving your existing applications and services to avoid relying on this feature. You can read more on the topic in the *documentation*.

Causality tracking

An obsolete early access feature to enable causality tracking, related to preview multi-domain, was removed. If you enabled it, you need to remove the following config lines, as they will not compile anymore:

```
participants.participant.init.parameters.unsafe-enable-causality-tracking = true
participants.participant.parameters.enable-causality-tracking = true
```

Besu and Fabric drivers

In order to allow for independent updates of the different components, we have moved the drivers into a separate jar, which needs to be loaded into a separate classpath. As a result, deployments that use Fabric or Besu need to additionally download the jar and place it in the appropriate directory. Please *consult the installation documentation* on how to obtain this additional jar.

Removal of deploy_sequencer_contract

The command `deploy_sequencer_contract` has been removed and exchanged with a deployment through genesis block in examples. The `deploy_sequencer_contract`, while convenient, is ill-suited for any production environment and can cause more damage than harm. The deployment of a sequencing contract should only happen once on the blockchain; however, adding deployment as part of the bootstrapping script would cause a redeployment each time bootstrapping is done.

Ledger API error codes

The error codes and metadata of gRPC errors returned as part of failed command interpretation from the Ledger API have been updated to include more information. Previously, most errors from the Daml engine would be given as either `GenericInterpretationError` or `InvalidArgumentInterpretationError`. They now all have their own codes and encode relevant information in the gRPC Status metadata. Specific error changes are as follows: * `GenericInterpretationError` (Code: `DAML_INTERPRETATION_ERROR`) with gRPC status `FAILED_PRECONDITION` is now split into:

`DisclosedContractKeyHashingError` (Code: `DISCLOSED_CONTRACT_KEY_HASHING_ERROR`) with gRPC status `FAILED_PRECONDITION`
`UnhandledException` (Code: `UNHANDLED_EXCEPTION`) with gRPC status `FAILED_PRECONDITION`
`InterpretationUserError` (Code: `INTERPRETATION_USER_ERROR`) with gRPC status `FAILED_PRECONDITION`
`TemplatePreconditionViolated` (Code: `TEMPLATE_PRECONDITION_VIOLATED`) with gRPC status `INVALID_ARGUMENT`

`InvalidArgumentInterpretationError` (Code: `DAML_INTERPRETER_INVALID_ARGUMENT`) with gRPC status `INVALID_ARGUMENT` is now split into:

- `CreateEmptyContractKeyMaintainers` (Code: `CREATE_EMPTY_CONTRACT_KEY_MAINTAINERS`) with gRPC status `INVALID_ARGUMENT`
- `FetchEmptyContractKeyMaintainers` (Code: `FETCH_EMPTY_CONTRACT_KEY_MAINTAINERS`) with gRPC status `INVALID_ARGUMENT`
- `WronglyTypedContract` (Code: `WRONGLY_TYPED_CONTRACT`) with gRPC status `FAILED_PRECONDITION`
- `ContractDoesNotImplementInterface` (Code: `CONTRACT_DOES_NOT_IMPLEMENT_INTERFACE`) with gRPC status `INVALID_ARGUMENT`
- `ContractDoesNotImplementRequiringInterface` (Code: `CONTRACT_DOES_NOT_IMPLEMENT_REQUIRING_INTERFACE`) with gRPC status `INVALID_ARGUMENT`
- `NonComparableValues` (Code: `NON_COMPARABLE_VALUES`) with gRPC status `INVALID_ARGUMENT`
- `ContractIdInContractKey` (Code: `CONTRACT_ID_IN_CONTRACT_KEY`) with gRPC status `INVALID_ARGUMENT`

- `ContractIdComparability` (Code: `CONTRACT_ID_COMPARABILITY`) with gRPC status `INVALID_ARGUMENT`
- `InterpretationDevError` (Code: `INTERPRETATION_DEV_ERROR`) with gRPC status `FAILED_PRECONDITION`

The `ContractKeyNotVisible` error (previously encapsulated by `GenericInterpretationError`) is now transformed into a `ContractKeyNotFound` to avoid information leaking.

Upgrade to Release 2.5

Version 2.5 will slightly extend the database schema used. Therefore, you will have to perform the database migration steps.

Some configuration arguments have changed. While rewrite rules are in-place for backwards compatibility, we recommend that you test your configuration prior to upgrading and update the settings to avoid using deprecated flags.

IMPORTANT: Existing domains and domain managers need to be reconfigured to keep on working. It is important that before attempting the binary upgrade, you configure the currently used protocol version explicitly:

```
canton.domains.mydomain.init.domain-parameters.protocol-version = 3
```

Nodes persist the static domain parameters used during initialization now. Version 2.5 is the last version that will require this explicit configuration setting during upgrading.

If you started the domain node accidentally before changing your configuration, your participants won't be able to reconnect to the domain, as they will fail with a message like:

> DOMAIN_PARAMETERS_CHANGED(9,d5dfa5ce): The domain parameters have changed

To recover from this, you need to force a reset of the stored static domain parameters using:

```
canton.domains.mydomain.init.domain-parameters.protocol-version = 3
canton.domains.mydomain.init.domain-parameters.reset-stored-static-config = yes
```

In order to benefit from protocol version 4, you will have to *upgrade the domain accordingly*.

Upgrade to Release 2.4

Version 2.4 will slightly extend the database schema used. Therefore, you will have to perform the database migration steps.

There have been a few consistency improvements to some console commands. In particular, we have renamed a few of the arguments and changed some of their types. As we have included automatic conversion and the change only affects special arguments (mainly timeouts), your script should still work. However, we recommend that you test your scripts for compilation issues. Please check the detailed release notes on the specific changes and their impact.

There was no change to the protocol. Participants / domains running 2.3 can also run 2.4, as both versions use the same protocol version.

Upgrade to Release 2.3

Version 2.3 will slightly extend the database schema used. Therefore, you will have to perform the database migration steps.

Furthermore, the Canton binary with version 2.3 has introduced a new protocol version 3, and deprecated the previous protocol version 2. In order to keep a node operational that is using protocol version 2, you need to turn on support for the deprecated protocol version.

On the participant, you need to turn on support for deprecated protocols explicitly:

```
canton.participants.myparticipant.parameters.minimum-protocol-version = 2.0.0
```

The default setting have changed to use protocol 3, while existing domains run protocol 2. Therefore, if you upgrade the binary on domain and domain manager nodes, you need to explicitly set the protocol version as follows:

```
canton.domains.mydomain.init.domain-parameters.protocol-version = 2.0.0
```

You cannot upgrade the protocol of a deployed domain! You need to keep it running with the existing protocol. Please follow the protocol upgrade guide to learn how to introduce a new protocol version.

1.31.2.2 Change the Canton Protocol Version

The Canton protocol is defined by the semantics and the wire-format used by the nodes to communicate to each other. In order to process transactions, all nodes must be able to understand and speak the same protocol.

Therefore, a new protocol can be introduced only once all nodes have been upgraded to a binary that can run the version.

Upgrade the Domain to a new Protocol Version

A domain is tied to a protocol version. This protocol version is configured when the domain is initialized and cannot be changed afterwards. Therefore, **you can not upgrade the protocol version of a domain**. Instead, you deploy a new domain side by side of the old domain process.

This applies to all domain members, be it sequencer, mediator or topology manager.

Please note that currently, the domain-id cannot be preserved during upgrades. The new domain must have a different domain-id due to the fact that the participant internally is associating a domain connection with a domain-id, and that association must be unique.

Therefore, the protocol upgrade process boils down to:

> Deploy a new domain next to the old domain. Ensure that the new domain is using the desired protocol version. Ensure that you are using different databases (or at least different schemas in the same database), channel names, smart contract addresses etc. It must be a completely separate domain (albeit you can reuse your DLT backend as long as you use different sequencer contract addresses or Fabric channels).
> Instruct the participants individually using the hard domain migration to use the new domain.

Note: to use the same database with different schemas for the old and the new domain, set the *currentSchema* either in the JDBC URL or as a parameter in *storage.config.properties*.

Hard Domain Connection Upgrade

A hard domain connection upgrade can be performed using the *respective migration command*. Again, please ensure that you have appropriate backups in place and that you have tested this procedure before applying it to your production system. You will have to enable these commands using a special config switch:

```
canton.features.enable-repair-commands=yes
```

The process of a hard migration is quite straightforward. Assuming that we have several participants, all connected to a domain named `olddomain`, then ensure that there are no pending transactions. You can do that by either controlling your applications, or by *setting the resource limits* to 0 on all participants:

```
@ participant.resources.set_resource_limits(ResourceLimits(Some(0), Some(0)))
```

This will reject all commands and finish processing the pending commands. Once you are sure that your participant node is idle, disconnect the participant node from the old domain connection:

```
@ participant.domains.disconnect("olddomain")
```

Test that the domain is disconnected by checking the list of active connections:

```
@ participant.domains.list_connected()
res3: Seq[ListConnectedDomainsResult] = Vector()
```

This is now a good time to perform a backup of the database before proceeding:

```
CREATE DATABASE newdb WITH TEMPLATE originaldb OWNER dbuser;
```

Next, we want to run the migration step. For this, we need to run the `repair.migrate_domain` command. The command expects two input arguments: The alias of the source domain and a domain connection configuration describing the new domain.

In order to build a domain connection config, we can just type

```
@ val config = DomainConnectionConfig("newdomain", GrpcSequencerConnection.
↪tryCreate("https://127.0.0.1:5018"))
config : DomainConnectionConfig = DomainConnectionConfig(
  domain = Domain 'newdomain',
  sequencerConnections = Sequencer 'DefaultSequencer' -> GrpcSequencerConnection(
    endpoints = https://127.0.0.1:5018,
    transportSecurity = true,
..
```

where the URL should obviously point to the correct domain. If you are testing the upgrade process locally in a single Canton process using a target domain named `newdomain` (which is what we are doing in this example here), you can grab the connection details using

```
@ val config = DomainConnectionConfig("newdomain", newdomain.sequencerConnection)
config : DomainConnectionConfig = DomainConnectionConfig(
  domain = Domain 'newdomain',
  sequencerConnections = Sequencer 'DefaultSequencer' -> GrpcSequencerConnection(
    endpoints = http://127.0.0.1:30154,
    transportSecurity = false,
..
```

Now, using this configuration object, we can trigger the hard domain connection migration using

```
@ participant.repair.migrate_domain("olddomain", config)
```

This command will register the new domain and re-associate the contracts tied to `olddomain` to the new domain.

Once all participants have performed the migration, they can reconnect to the domain

```
@ participant.domains.reconnect_all()
```

Now, the new domain should be connected:

```
@ participant.domains.list_connected()
res8: Seq[ListConnectedDomainsResult] = Vector(
  ListConnectedDomainsResult(
    domainAlias = Domain 'newdomain',
    domainId = newdomain::1220b732056e...,
    healthy = true
  )
)
```

As we've previously set the resource limits to 0, we need to reset this back

```
@ participant.resources.set_resource_limits(ResourceLimits(None, None))
```

Finally, we can test that the participant can process a transaction by running a ping on the new domain

```
@ participant.health.ping(participant)
res10: Duration = 890 milliseconds
```

Note: Note that currently, the hard migration is the only supported way to migrate a production system. This is due to the fact that unique contract keys are restricted to a single domain.

While the domain migration command is mainly used for upgrading, it can also be used to recover contracts associated to a broken domain. Domain migrations can be performed back and forth, allowing to roll back in case of issues.

After the upgrade, the participants may report mismatch between commitments during the first commitment exchange, as they might have performed the migration at slightly different times. The warning should eventually stop once all participants are back up and connected.

Expected Performance

Performance-wise, we can note the following: when we migrate contracts, we write directly into the respective event logs. This means that on the source domain, we insert transfer-out, while we write a transfer-in and the contract into the target domain. Writing this information is substantially faster than any kind of transaction processing (several thousand migrations per second on a single cpu / 16 core test server). However, with very large datasets, the process can still take quite some time. Therefore, we advise to measure the time the migration takes during the upgrade test in order to understand the necessary downtime required for the migration.

Furthermore, upon reconnect, the participant needs to recompute the new set of commitments. This can take a while for large numbers of contracts.

Soft Domain Connection Upgrade

Note: The soft domain connection upgrade is currently only supported as an alpha feature.

The hard domain connection upgrade requires coordination among all participants in a network. The soft domain connection upgrade is operationally much simpler, and can be leveraged using multi-domain support (which exists as a pre-alpha feature only for now). By turning off non-unique contract keys, participants can connect to multiple domains and transfer contracts between domains. This allows us to avoid using the `repair.migrate_domain` step.

Assuming the same setup as before, where the participant is connected to the old domain, we can just connect it to the new domain

```
@ participant.domains.connect_local(newdomain)
```

Give the new connection precedence over the old connection by changing the `priority` flag of the new domain connection:

```
@ participant.domains.modify("newdomain", _.copy(priority=10))
```

You can check the priority settings of the domains using

```
@ participant.domains.list_registered().map { case (c,_) => (c.domain, c.
↪priority) }
res3: Seq[(com.digitalasset.canton.DomainAlias, Int)] = Vector((Domain 'newdomain
↪', 10), (Domain 'olddomain', 0))
```

Existing contracts will not automatically move over to the new domain. The domain router will pick the domain by minimizing the number of transfers and the priority. Therefore, most contracts will remain on the old domain without additional action. However, by using the *transfer command*, contracts can be moved over to the new domain one by one, such that eventually, all contracts are associated with the new domain, allowing the old domain to be decommissioned and turned off.

The soft upgrade path provides a smooth user experience that does not require a hard migration of the domain connection synchronised across all participants. Instead, participants upgrade individually, whenever they are ready, allowing them to reverse the process if needed.

1.31.3 Auth0 Example Configuration

This section describes a minimal example configuration of the trigger service with authorization enabled using Auth0 as the OAuth 2.0 provider together with the OAuth 2.0 middleware included in Daml. It uses the sandbox as the Daml ledger.

1.31.3.1 Configure Auth0

Sign up for an account on Auth0 to follow this guide.

Create an API

First, create a new API on the Auth0 API dashboard. This will represent the Daml ledger API and controls properties of access tokens issued for the ledger API.

Enter the name of the API, e.g. `ex-daml-api`.
Enter the API identifier: `https://daml.com/ledger-api`.
Select the signing algorithm `RS256`.
Press the create button.

Enter the settings of the newly created API.

Allow offline access in the access settings section to enable issuance of refresh tokens.

Create an Application

Create a new native application. This will represent the OAuth 2.0 middleware.

Enter the name of the application, e.g. `ex-daml-auth-middleware`.
Choose the application type native .
Press the create button.

Enter the settings of the newly created application.

Configure the allowed callback URLs: `http://localhost:5000/auth/cb`.
This is the URL to the callback endpoint of the auth middleware, in this case through the reverse proxy.
Take note of the Client ID and Client Secret displayed in the Basic Information section.
Take note of the following URLs in the Endpoints tab of the advanced settings:

- OAuth Authorization URL,
- OAuth Token URL, and
- JSON Web Key Set.

Create a Rule

Create a new rule. This will define user privileges, the mapping from scopes to ledger claims, and construct the access token.

Note, for simplicity this rule will grant access to any claims to any user. In a real setup the rule will need to validate whether the user is authorized to access the requested claims. Rules can be used to implement custom authorization policies.

This rule will define a one-to-one mapping between scopes and Daml ledger claims, this is compatible with the default request templates that are built into the OAuth 2.0 middleware.

Enter the name of the rule, e.g. `ex-daml-token`.
Enter the following script:

```
function (user, context, callback) {
  // NOTE change the ledger ID to match your deployment.
  const ledgerId = 'daml-auth0-example-ledger';
  const apiId = 'https://daml.com/ledger-api';

  const query = context.request.query;

  // Only handle ledger-api audience.
  const audience = query && query.audience || "";
  if (audience !== apiId) {
    return callback(null, user, context);
  }

  // Determine requested claims.
  var admin = false;
  var readAs = [];
  var actAs = [];
  var applicationId = null;
  const scope = (query && query.scope || "").split(" ");
  scope.forEach(s => {
    if (s === "admin") {
      admin = true;
    } else if (s.startsWith("readAs:")) {
      readAs.push(s.slice(7));
    } else if (s.startsWith("actAs:")) {
      actAs.push(s.slice(6));
    } else if (s.startsWith("applicationId:")) {
      applicationId = s.slice(14);
    }
  });

  // Construct access token.
  context.accessToken[apiId] = {
    "ledgerId": ledgerId,
    "actAs": actAs,
    "readAs": readAs,
    "admin": admin
  };
  if (applicationId) {
    context.accessToken[apiId].applicationId = applicationId;
  }
```

(continues on next page)

(continued from previous page)

```
  return callback(null, user, context);
}
```

You can use the Real-time Webtask Logs extension to view any `console.log` output generated by your rule during the processing of authorization requests.

Create a User

Create a new user.

Enter an email address, e.g. `alice@example.com`.
Enter a secure password.
Remember the credentials.
Choose the Username-Password-Authentication connection.
Press the create button.

Enter the details page of the newly created user.

Edit the email address.
Press Set email as verified .
Press save .

1.31.3.2 Start Daml

Next, configure the relevant Daml components to use Auth0 as the IAM.

Sandbox

Start the sandbox using the following command. Replace `JSON_Web_Key_Set` by the corresponding URL found in the application settings and make sure that the ledger ID matches the one in the Auth0 rule.

```
daml sandbox \
  --address localhost \
  --port 6865 \
  --ledgerid daml-auth0-example-ledger \
  --wall-clock-time \
  --auth-jwt-rs256-jwks "JSON_Web_Key_Set"
```

OAuth 2.0 Middleware

Start the auth middleware using the following command. Replace the client identifier and URL placeholders by the corresponding values found in the application settings and make sure that the callback URL matches the allowed callback URL in the application settings. The `--callback` flag defines the middleware's callback URL as exposed through the reverse proxy.

```
DAML_CLIENT_ID="Client_ID" \
DAML_CLIENT_SECRET="Client_Secret" \
daml oauth2-middleware \
```

(continues on next page)

(continued from previous page)

```
  --address localhost \
  --http-port 3000 \
  --oauth-auth "OAuth_Authorization_URL" \
  --oauth-token "OAuth_Token_URL" \
  --auth-jwt-rs256-jwks "JSON_Web_Key_Set" \
  --callback http://localhost:5000/auth/cb
```

Trigger Service

Start the trigger service using the following command. The `--auth` flag defines the middleware's URL prefix as exposed through the reverse proxy, similarly the `--auth-callback` flag defines the trigger service's callback URL as exposed through the reverse proxy.

```
daml trigger-service \
  --address localhost \
  --http-port 4000 \
  --ledger-host localhost \
  --ledger-port 6865 \
  --auth http://localhost:5000/auth \
  --auth-callback http://localhost:5000/trigger/cb
```

1.31.3.3 Configure Web Server

This guide uses Nginx as a reverse proxy and web server.

Configure nginx using the following snippet:

```
http {
  server {
    listen 5000;
    server_name localhost;
    root html;

    location /auth/ {
      proxy_pass http://localhost:3000/;
    }

    location /trigger/ {
      proxy_pass http://localhost:4000/;
    }
  }
}
```

This exposes the auth middleware under the URL `http://localhost:3000/` and the trigger service under the URL `http://localhost:4000/`.
Add the following `index.html` to your web root:

```
<!DOCTYPE html>
<html>
  <body>
    <button onclick="listTriggers()">list triggers</button>
  </body>
```

(continues on next page)

(continued from previous page)

```
  <script>
    async function listTriggers() {
      // The rule defined above accepts all claims for all users.
      // So, we can always access claims to the party Alice.
      const resp = await fetch("http://localhost:5000/trigger/v1/triggers?
↪party=Alice");
      if (resp.status === 401) {
        const challenge = await resp.json();
        console.log(`Unauthorized ${JSON.stringify(challenge)}`);
        var loginUrl = new URL(challenge.login);
        loginUrl.searchParams.append("redirect_uri", window.location.href);
        window.location.replace(loginUrl.href);
      } else {
        const body = await resp.text();
        console.log(`(${resp.status}) ${body}`);
      }
    }
  </script>
</html>
```

This defines a very simple web site with a single button that will request the list of Alice's running triggers from the trigger service. If the user is authorized it will print the list to the JavaScript console, otherwise it will redirect to auth middleware's login endpoint to obtain authorization.

1.31.3.4 Test the Setup

Use the following commands to determine if the OAuth 2.0 middleware and trigger service are running and available through the reverse proxy.

```
$ curl http://localhost:5000/auth/livez
{"status":"pass"}
$ curl http://localhost:5000/trigger/livez
{"status":"pass"}
```

Direct your web browser to the URL `http://localhost:5000`. It should display the test page with the single list triggers button defined above.

Open the JavaScript console.
Press the list triggers button.
An Unauthorized message should appear in the console and you should be redirected to the auth0 login page.
Login with the credentials of the auth0 user that you created before.
The browser should be redirected to the test page.
Click the button again. This time a message like the following should appear in the console.

```
(200) {"result":{"triggerIds":[]},"status":200}
```

1.31.4 Security

1.31.4.1 Cryptographic Key Usage

This section covers the generation and usage of cryptographic keys in the Canton nodes. It assumes that the configuration sets *auto-init = true* which leads to the generation of the default keys on a node's startup.

The scope of cryptographic keys covers all Canton-protocol specific keys, private keys for TLS, as well as additional keys required for the domain integrations, e.g., with Besu.

Supported Cryptographic Schemes in Canton

Within Canton we use the cryptographic primitives of signing, symmetric and asymmetric encryption, and MAC with the following supported schemes (*D = default*, *S = supported*, *P = partially supported* for instance just signature verification but no signing with a private key, and */ = not supported*):

Crypto Provider	Tink	JCE	KMS
Signing			
Ed25519	D	D	P
ECDSA P-256	S	S	D
ECDSA P-384	S	S	S
Symmetric Encryption			
AES128-GCM	D	D	D
Asymmetric Encryption			
ECIES on P-256 with HMAC-SHA256 and AES128-GCM	D	D	/
ECIES on P-256 with HMAC-SHA256 and AES128-CBC	/	S	/
RSA 2048 with OAEP using SHA-256	/	S	D
MAC			
HMAC with SHA-256	D	D	D

Key Generation and Storage

Keys can either be generated in the node and stored in the node's primary storage or generated and stored by an external key management system (KMS). We currently support a version of Canton that can use a KMS to either: (a) *protect Canton's private keys at rest* or (b) *generate and store the private keys itself*. This version is available only as part of Daml Enterprise.

You can find more background information on this key management feature in *Secure Cryptographic Private Key Storage*. See *Protect Private Keys With Envelope Encryption and a Key Management Service* if you wish to know how Canton can protect private keys whilst they remain internally stored in Canton using a KMS, or *Externalize Private Keys With a Key Management Service* for more details on how Canton can enable private keys to be generated and stored by an external KMS.

The following section *Key Management Service Setup* describes how to enable KMS support in Canton and how to setup each of these two modes of operation.

Public Key Distribution using Topology Management

The public keys of the corresponding key pairs that are used for signing and asymmetric encryption within Canton are distributed using Canton's Topology Management. Specifically, signing and asymmetric encryption public keys are distributed using *OwnerToKeyMapping* transactions, which associate a node with a public key for either signing or encryption, and *NamespaceDelegation* for namespace signing public keys.

See *Topology Transactions* for details on the specific topology transactions in use.

Common Node Keys

Each node provides an Admin API for administrative purposes, which is secured using TLS.

The node reads the private key for the TLS server certificate from a file at startup.

Participant Node Keys

Participant Namespace Signing Key

A Canton participant node spans its own identity namespace, for instance for its own id and the Daml parties allocated on the participant node. The namespace is the hash of the public key of the participant namespace signing key.

The private key is used to sign and thereby authorize all topology transactions for this namespace and this participant, including the following transactions:

Root *NamespaceDelegation* for the new identity namespace of the participant
OwnerToKeyMapping for all the public keys that the participant will generate and use (these keys will be explained in the follow-up sections)
PartyToParticipant for the parties allocated on this participant
VettedPackages for the packages that have been vetted by this participant

Signing Key

In addition to the topology signing key, a participant node will generate another signing key pair that is used for the Canton transaction protocol in the following cases:

Sequencer Authentication: Signing the nonce generated by the sequencer as part of its challenge-response authentication protocol. The sequencer verifies the signature with the public key registered for the member in the topology state.
Transaction Protocol - The Merkle tree root hash of confirmation requests is signed for a top-level view. - The confirmation responses sent to the mediator are signed as a whole. - The Merkle tree root hash of transfer-in and transfer-out messages is signed.
Pruning: Signing of ACS commitments.

Participant Encryption Key

In addition to a signing key pair, a participant node also generates a key pair for encryption based on an asymmetric encryption scheme. A transaction payload is encrypted for a recipient based on the recipient's public encryption key that is part of the topology state.

See the next section on how a transaction is encrypted using an ephemeral symmetric key.

View Encryption Key

A transaction is composed of multiple views due to sub-transaction privacy. Instead of duplicating each view by directly encrypting the view for each recipient using their participant encryption public key, Canton derives a symmetric key for each view to encrypt that view. The key is derived using a HKDF from a secure seed that is only stored encrypted under the public encryption key of a participants. Thereby, only the encrypted seed is duplicated but not a view.

Ledger API TLS Key

The private key for the TLS server certificate is provided as a file, which can optionally be encrypted and the symmetric decryption key is fetched from a given URL.

Domain Topology Manager Keys

Domain Namespace Signing Key

The domain topology manager governs the namespace of the domain and has a signing key pair for the namespace. The hash of the public key forms the namespace and all entities in the domain (mediator, sequencer, the topology manager itself) may have identities under the domain namespace.

The domain topology manager signs and thereby authorizes the following topology transactions:

NamespaceDelegation to register the namespace public key for the new namespace
OwnerToKeyMapping to register both its own signing public key (see next section) and the signing public keys of the other domain entities as part of the domain onboarding
ParticipantState to enable a new participant on the domain
MediatorDomainState to enable a new mediator on the domain

Signing Key

The domain topology manager is not part of the Canton transaction protocol, but it receives topology transactions via the sequencer. Therefore, in addition to the domain namespace, the domain topology manager has a signing key pair, which is registered in the topology state for the topology manager. This signing key is used to perform the challenge-response protocol of the sequencer.

Sequencer Node Keys

Signing Key

The sequencer has a signing key pair that is used to sign all events the sequencer sends to a subscriber.

Ethereum Sequencer

The Ethereum-based sequencer is a client of a Besu node and additional keys are used in this deployment:

TLS client certificate and private key to authenticate towards a Besu node if mutual authentication is configured.
A Wallet (in BIP-39 or UTC / JSON format), which contains or will result in a signing key pair for Ethereum transactions.

Fabric Sequencer

The Fabric-based sequencer is a Fabric application connecting to an organization's peer node and the following additional keys are required:

TLS client certificate and private key to authenticate towards a Fabric peer node if mutual authentication is required.
The client identity's certificate and private key.

Public API TLS Key

The private key for the TLS server certificate is provided as a file.

Mediator Node Keys

Signing Key

The mediator node is part of the Canton transaction protocol and uses a signing key pair for the following:

Sequencer Authentication: Signing of the challenge as part of the sequencer challenge-response protocol.
Signing of transaction results, transfer results, and rejections of malformed mediator requests.

Domain Node Keys

The domain node embeds a sequencer, mediator, and domain topology manager. The set of keys remains the same as for the individual nodes.

Canton Console Keys

When the Canton console runs separate from the node and mutual authentication is configured on the Admin API, then the console requires a TLS client certificate and corresponding private key as a file.

1.31.4.2 Cryptographic Key Management

Rotating Canton Node Keys

Canton supports rotation of node keys (signing and encryption) during live operation through its topology management. In order to ensure continuous operation, the new key is added first and then the previous key is removed.

For participant nodes, domain nodes, and domain topology managers, the nodes can rotate their keys directly using their own identity manager with the following command for example:

```
participant1.keys.secret.rotate_node_keys()
```

On a participant node both the signing and encryption key pairs are rotated. On a domain and domain manager node only the signing key pair, because they do not have a encryption key pair. Identity namespace root or intermediate keys are not rotated with this command, see below for commands on namespace key management.

For sequencer and mediator nodes that are part of a domain, the domain topology manager authorizes the key rotation and a reference needs to be passed in to the command, for example:

```
domainManager1.keys.secret.rotate_node_keys()
sequencer1.keys.secret.rotate_node_keys(domainManager1)
mediator1.keys.secret.rotate_node_keys(domainManager1)
```

We can also individually rotate a key by running the following command for example:

```
participant1.keys.secret.rotate_node_keys()
```

A fingerprint of a key can be retrieved from the list of public keys:

```
participant2.keys.secret
  .list()
```

Namespace Intermediate Key Management

Relying on the namespace root key to authorize topology transactions for the namespace is problematic because we cannot rotate the root key without losing the namespace. Instead we can create intermediate keys for the namespace, similar to an intermediate certificate authority, in the following way:

```
// create a new namespace intermediate key
val intermediateKey = identityManager.keys.secret.generate_signing_key()

// Create a namespace delegation for the intermediate key with the namespace root
↪key
identityManager.topology.namespace_delegations.authorize(
  TopologyChangeOp.Add,
  rootKey.fingerprint,
  intermediateKey.fingerprint,
)
```

We can rotate an intermediate key by creating a new one and renewing the existing topology transactions that have been authorized with the previous intermediate key. First the new intermediate key has to be created in the same way as the initial intermediate key. To rotate the intermediate key and renew existing topology transactions:

```
// Renew all active topology transactions that have been authorized by the
↪previous intermediate key with the new intermediate key
identityManager.topology.all.renew(intermediateKey.fingerprint,
↪newIntermediateKey.fingerprint)

// Remove the previous intermediate key
identityManager.topology.namespace_delegations.authorize(
  TopologyChangeOp.Remove,
  rootKey.fingerprint,
  intermediateKey.fingerprint,
)
```

Moving the Namespace Secret Key to Offline Storage

An identity is ultimately bound to a particular secret key. Owning that secret key gives full authority over the entire namespace. From a security standpoint, it is therefore critical to keep the namespace secret key confidential. This can be achieved by moving the key off the node for offline storage. The identity management system can still be used by creating a new key and an appropriate intermediate certificate. The following steps illustrate how:

```
// fingerprint of namespace giving key
val participantId = participant1.id
val namespace = participantId.uid.namespace.fingerprint

// create new key
val name = "new-identity-key"
val fingerprint = participant1.keys.secret.generate_signing_key(name = name).
↪fingerprint

// create an intermediate certificate authority through a namespace delegation
```

(continues on next page)

(continued from previous page)

```
// we do this by adding a new namespace delegation for the newly generated key
// and we sign this using the root namespace key
participant1.topology.namespace_delegations.authorize(
  TopologyChangeOp.Add,
  namespace,
  fingerprint,
  signedBy = Some(namespace),
)

// export namespace key to file for offline storage, in this example, it's a
↪temporary file
better.files.File.usingTemporaryFile("namespace", ".key") { privateKeyFile =>
  participant1.keys.secret.download_to(namespace, privateKeyFile.toString)

  // delete namespace key (very dangerous ...)
  participant1.keys.secret.delete(namespace, force = true)
```

When the root namespace key is required, it can be imported again on the original node or on another, using the following steps:

```
// import it back wherever needed
other.keys.secret.upload(privateKeyFile.toString, Some("newly-imported-identity-
↪key"))
```

Identifier Delegation Key Management

Identifier delegations work similar to namespace delegations, however a key is only allowed to operate on a specific identity and not an entire namespace (cf. *Topology Transactions*).

Therefore the key management for identifier delegations also works the same way as for namespace delegations, where all the topology transactions authorized by the previous identifier delegation key have to be renewed.

Key Management Service Setup

Important: This feature is only available in Canton Enterprise

---

Canton supports using a Key Management Service (KMS) to increase security of stored private keys.

The **first way** to do this is by (1) storing Canton's private keys in a node's database in an encrypted form and then (2) upon startup the KMS decrypts these keys for use by Canton. The unencrypted keys are stored in memory so this approach increases security without impacting performance. This is a common approach used by KMS vendors; using a symmetric encryption key, called the *KMS wrapper key*, to encrypt and decrypt the stored, private keys.

The **second way** is to directly use a KMS to generate and store Canton's private keys and then use its API to securely sign an decrypt messages. A Canton node still stores the corresponding public keys in its stores so that it can verify signatures and encrypt messages without having to rely on the KMS.

The KMS integration is currently enabled for *Amazon Web Services (AWS) KMS* and *Google Cloud Provider (GCP) KMS* in Canton Enterprise.

Running Canton with a KMS

KMS support can be enabled for a new installation (i.e., during the node bootstrap) or for an existing deployment. When the KMS is enabled after a node has been running, the keys are (a) encrypted and stored in this encrypted form in the Canton node's database, or (b) transparently replaced by external KMS keys. For scenario (a) this process is done transparently, while in (b) *a node needs to be migrated* if the key schemes being used do not match the current supported keys for KMS.

Note: In scenario (a), the KMS keys used to encrypt the private keys need to live as long as the Canton database backups, so care must be taken when deleting database backup files or KMS keys. Otherwise, a Canton node restored from a database backup may try to decrypt the private keys with a *KMS wrapper key* that was previously deleted.

Canton Configuration of a KMS

Like other Canton capabilities, KMS integration is enabled within a Canton node's configuration file. A KMS for AWS or GCP is configured in the following way:

`type` specifies which KMS to use.

```
canton.participants.participant1.crypto.kms {
        type = aws
        region = us-east-1
        multi-region-key = false # optional, default is false
        audit-logging = false # optional, default is false
}
```

Specific to AWS:

`region` specifies which region the AWS KMS is bound to.
`multi-region-key` flag enables the replication of keys generated by the KMS. With replication turned on, the operator can replicate a key from one region to another (Note: replication of a key is not done automatically by Canton) and change the region configured in Canton at a later point in time without any other key rotation required. **The standard single-region approach is applicable for most scenarios**.

```
canton.participants.participant2.crypto.gcp {
    type = gcp
    location-id = us-east1,
    project-id = gcp-kms-testing,
    keyRing-id = canton-test-keys-2023,
}
```

Specific to GCP:

`location-id` specifies which region the GCP KMS is bound to.
`project-id` specifies which project are we binding to.
`keyRingId` specifies the keyring to use. Contrary to AWS, multi region keys are enabled for an entire keyring. Therefore, the KMS operator is responsible for setting the keyring correctly depending on the systems' needs.

Configure AWS Credentials and Permissions

When using a KMS to envelope encrypt the private keys stored in Canton, it needs to be configured with the following list of authorized actions (i.e. IAM permissions):

AWS	GCP
kms:CreateKey	*cloudkms.cryptoKeyVersions.create*
kms:TagResource	-
kms:Encrypt	*cloudkms.cryptoKeyVersions.useToEncrypt*
kms:Decrypt	*cloudkms.cryptoKeyVersions.useToDecrypt*
kms:DescribeKey	*cloudkms.cryptoKeys.get*

When we rely on a KMS to generate, store, and manage the necessary private keys, it must be configured with the following list of authorized actions:

AWS	GCP
kms:CreateKey	*cloudkms.cryptoKeyVersions.create*
kms:TagResource	-
kms:Decrypt	*cloudkms.cryptoKeyVersions.useToDecrypt*
kms:Sign	*cloudkms.cryptoKeyVersions.useToEncrypt*
kms:DescribeKey	*cloudkms.cryptoKeyVersions.useToSign*
kms:GetPublicKey	*cloudkms.cryptoKeyVersions.viewPublicKey*

If you plan to use cross-account key usage then the permission for key rotation in Canton, namely *kms:CreateKey*, does not have to be configured as it does not apply in that use case.

To make the API calls to the AWS KMS, Canton uses the standard AWS credential access. For example, the standard environment variables of *AWS_ACCESS_KEY_ID* and *AWS_SECRET_ACCESS_KEY* can be used. Alternatively, you can specify an AWS profile file (e.g. use a temporary access profile credentials - *sts*).

For GCP, Canton uses a GCP service account. For example, the standard environment variable *GOOGLE_APPLICATION_CREDENTIALS* can be used after setting up a local Application Default Credentials (ADC) file for our service account.

The protection and rotation of the credentials for AWS or GCP are the responsibility of the node operator.

Canton Configuration for Encrypted Private Key Storage

In the example below the encrypted private key storage integration is enabled for a participant node (called `participant1`). The same applies for any other node, such as a sync domain manager, a mediator, or a sequencer.

The most important setting that enables an encrypted private key storage using a KMS is "type = kms". This is shown below. If this is not specified, Canton stores the keys using its default approach, which is in unencrypted form.

```
canton.participants.participant1.crypto.private-key-store.encryption.type = kms
```

There are two ways to choose the KMS wrapper key: (1) use an already existing KMS key or; (2) let Canton generate one. To use an already existing KMS key, you must specify its identifier. For example, for AWS KMS this can be one of the following:

Key id: *1234abcd-12ab-34cd-56ef-1234567890ab*
Key ARN (Amazon Resource Name): *arn:aws:kms:us-east-1:1234abcd-12ab-34cd-56ef-1234567890ab*
Key alias: *alias/test-key*

Please be aware that an AWS KMS key needs to be configured with the following settings:

Key specification: SYMMETRIC_DEFAULT
Key usage: ENCRYPT_DECRYPT

Similarly, for GCP KMS we can use:

Key name: *test-key*
Key RN (Resource Name): *projects/gcp-kms-testing/locations/us-east1/keyRings/canton-test-keys/cryptoKeys/t*

And your key needs to be configured with the following settings:

Key algorithm: GOOGLE_SYMMETRIC_ENCRYPTION
Key purpose: *ENCRYPT_DECRYPT*

If no `wrapper-key-id` is specified, Canton creates a symmetric key in the KMS. After subsequent restarts the operator does not need to specify the identifier for the newly created key; Canton stores the generated wrapper key id in the database.

An example with a pre-defined AWS KMS key is shown below:

```
canton.participants.participant1.crypto.private-key-store.encryption.wrapper-key-
↪id = alias/canton-kms-test-key
```

An example configuration that puts it all together is below:

```
canton.participants.participant1.crypto.private-key-store.encryption.type = kms
canton.participants.participant1.crypto.private-key-store.encryption.wrapper-key-
↪id = alias/canton-kms-test-key
canton.participants.participant1.crypto.kms {
        type = aws
        region = us-east-1
        multi-region-key = false
}
```

Revert Encrypted Private Key Storage

If you wish to change the encrypted private key store and revert back to using an unencrypted store, you must restart the nodes with an updated configuration that includes

```
canton.participants.participant1.crypto.private-key-store.encryption.reverted =
↪true # default is false
```

Warning: We strongly advise against this as it will force Canton to decrypt its private keys and store them in clear.

For subsequent restarts we recommend deleting all encrypted private key store configurations including the KMS store. We have forced the manual configuration of the *reverted* flag to prevent any unwanted decryption of the database (e.g. by unintentionally deleting the KMS configuration).

Manual wrapper key rotation

Currently AWS and GCP offer automatic KMS symmetric key rotation (yearly for AWS and user-defined for GCP). Canton extends this by enabling node administrators to manually rotate the KMS wrapper key using the following command:

```
participant1.keys.secret.rotate_wrapper_key(newWrapperKeyId)
```

You can optionally pass a wrapper key id to change to or let Canton generate a new key based on the current KMS configuration.

Note: Changing the key specification (e.g. enable multi region) during rotation is for now only possible with AWS, by updating the configuration before rotating the wrapper key.

Canton Configuration for External Key Storage and Usage

In the example below, we configure a Canton participant node (called `participant1`) to generate and store private keys in an external KMS. Besides the previously presented *KMS configuration* (in this example we use AWS, but GCP is set similarly) you only need to specify the correct crypto provider `kms` and ensure that the remaining nodes, in particular the connected domain, runs with the correct schemes:

```
canton.domains.da.crypto.provider = jce
canton.domains.da.init.domain-parameters.required-signing-key-schemes = [ ec-dsa-
↪p-256 ]
canton.domains.da.init.domain-parameters.required-encryption-key-schemes = [ rsa-
↪2048-oaep-sha-256 ]

canton.participants.participant1.crypto.provider = kms
canton.participants.participant1.crypto.kms {
        type = aws
        region = us-east-1
        multi-region-key = false # optional, default is false
}
```

Therefore, a node running with a `kms` provider is only ever able to communicate with other nodes running a `kms` or `jce` providers. Furthermore, the nodes have to be explicitly configured to use the KMS supported algorithms as the required algorithms.

AWS and GCP KMSs only support the *following cryptographic schemes*.

Note: You cannot mix an external private key storage configuration with an encrypted private key storage configuration. Currently if a node starts with a KMS as its provider it can no longer be reverted without a full reset of the node (i.e., re-generation of node identity and all keys).

Setup with Pre-Generated Keys

In the previous example, Canton creates its own keys on startup and initializes the identity of the nodes automatically. If the keys have already been generated in the KMS, we need to manually initialize the identity of the nodes by adding the following flag in the config:

```
<node>.init.auto-init = false
```

We then need to register the keys in Canton by running the key registration command on each node. For example for a participant we would run:

```
participant.keys.secret.register_kms_signing_key(namespaceKmsKeyId),
participant.keys.secret.register_kms_signing_key(signingKmsKeyId),
participant.keys.secret.register_kms_encryption_key(encryptionKmsKeyId),
```

where *xyzKmsKeyId* is the KMS identifier for a specific key (e.g. *KMS Key RN*). If we are using, for example, AWS cross account keys be aware that using the key id is not enough and we are required to register the key using its *ARN*.

Finally, we need to initialize our *domain* and *participants* using the previously registered keys.

Participant Node Migration to KMS Crypto Provider

To migrate an existing participant node connected to a domain with a non KMS-compatible provider and start using KMS external keys, we need to manually execute the following steps. The general idea is to replicate the old node into a *new one that uses a KMS provider and connects to a KMS-compatible domain* (e.g. running JCE with KMS supported encryption and signing keys).

First, we need to delegate the namespace of the old participant to the new participant:

```
val namespaceNew = participantNew.uid.namespace.fingerprint
val namespaceOld = participantOld.uid.namespace.fingerprint

val rootNamespaceDelegationOld = participantOld.topology.namespace_delegations
  .list(filterNamespace = namespaceOld.toProtoPrimitive)
  .head
  .context
  .serialized

val namespaceKeyNew = participantNew.keys.public.download(namespaceNew)
participantOld.keys.public.upload(namespaceKeyNew, Some("pNew-namespace-key"))

// Delegate namespace of old participant to new participant
val delegation = participantOld.topology.namespace_delegations.authorize(
  ops = TopologyChangeOp.Add,
  namespace = namespaceOld,
  authorizedKey = namespaceNew,
)

participantNew.topology.load_transaction(rootNamespaceDelegationOld)
participantNew.topology.load_transaction(delegation)
```

Secondly, we must recreate all parties of the old participant in the new participant:

```
val parties = participantOld.parties.list().map(_.party)

// Disconnect from new KMS-compatible domain to prepare migration of parties and
↪contracts
participantNew.domains.disconnect(kmsDomain)

parties.foreach { party =>
  participantNew.topology.party_to_participant_mappings
    .authorize(ops = TopologyChangeOp.Add, party = party, participant =
↪participantNew.id)
}
```

Finally, we need to transfer the active contracts of all the parties from the old participant to the new one and connect to the new domain:

```
val parties = participantOld.parties.list().map(_.party)

// Make sure domain and the old participant are quiet before exporting ACS
participantOld.domains.disconnect("acme")
acme.stop()

File.usingTemporaryFile("participantOld-acs", suffix = ".txt") { acsFile =>
  val acsFileName = acsFile.toString

  // Export from old participant
  participantOld.repair.download(
    parties = parties.toSet,
    outputFile = acsFileName,
    contractDomainRenames = Map(kmsDomainId -> newDomainId),
  )

  // Import to new participant
  participantNew.repair.upload(acsFileName)
}

// Kill/stop the old participant
participantOld.stop()

// Connect the new participant to a new domain
participantNew.domains.reconnect("da")
```

The end result is a new participant node with its keys stored and managed by a KMS connected to a domain that is able to communicate using the appropriate key schemes.

Manual KMS key rotation

Canton keys can still be manually rotated even if they are externally stored in a KMS. To do that we can use the same *standard rotate key commands* or, if we already have a KMS key to rotate to, run the following command:

```
val newSigningKey = participant1.keys.secret
  .rotate_kms_node_key(
    keyFingerprint,
    newKmsKeyId,
  )
```

Neither AWS or GCP offer automatic rotation of asymmetric keys so, unlike the wrapper key rotation, the node operator needs to be responsible for periodically rotating these keys.

Auditability

AWS and GCP provide tools to monitor KMS keys. For AWS to set automatic external logging, refer to the AWS official documentation. This includes instructions on how to set AWS Cloud Trail or Cloud Watch Alarms to keep track of usage of KMS keys or of performed crypto operations. For GCP you can refer to the GCP official documentation for information on logging. Errors resulting from the use of KMS keys are logged in Canton.

Logging

For further auditability, Canton can be configured to log every call made to the AWS KMS. To enable this feature, set the `audit-logging` field of the KMS configuration to `true`. By default, when using a file-based logging configuration, such logs will be written into the main canton log file. To write them to a dedicated log file, set the `KMS_LOG_FILE_NAME` environment variable or `--kms-log-file-name` CLI flag to the path of the file. These and other parameters can be configured using environment variables or CLI flags:

Table 3: KMS logging configuration

Environment variable	CLI Flag	Purpose	Default
KMS_LOG_FILE_NAME	–kms-log-file-name	Path to a dedicated KMS log file	not set
KMS_LOG_IMMEDIATE_FLUSH	–kms-log-immediate-flush	When true, logs will be immediately flushed to the KMS log file	true
KMS_LOG_FILE_ROLLING_PATTERN	–kms-log-file-rolling-pattern	Pattern to use when using the rolling file strategy to roll KMS log files	yyyy-MM-dd
KMS_LOG_FILE_HISTORY	–kms-log-file-history	Maximum number of KMS log files to keep when using the rolling file strategy	0 (i.e. no limit)

Sample of an AWS KMS audit log:

```
2023-09-12 15:44:54,426 [env-execution-context-27] INFO  c.d.c.c.k.a.a.
↪AwsRequestResponseLogger:participant=participant1
↪tid:40d47592f1bd50f37e6804fbdff404dd - Sending request [06cc259e220da647]:
↪DecryptRequest(CiphertextBlob=** Ciphertext placeholder **, KeyId=91c48ce4-ec80-
↪44c1-a219-fdd07f12f002, EncryptionAlgorithm=RSAES_OAEP_SHA_256) to https://kms.
↪us-east-1.amazonaws.com/
2023-09-12 15:44:54,538 [aws-java-sdk-NettyEventLoop-1-15] INFO  c.d.c.c.k.a.a.
↪AwsRequestResponseLogger:participant=participant1
↪tid:40d47592f1bd50f37e6804fbdff404dd - Received response [06cc259e220da647]:
↪[Aws-Id: 1836823c-bb8a-44bf-883d-f33d696bf84f] - DecryptResponse(Plaintext=**
↪Redacted plaintext placeholder **, KeyId=arn:aws:kms:us-east-1:724647588434:key/
↪91c48ce4-ec80-44c1-a219-fdd07f12f002, EncryptionAlgorithm=RSAES_OAEP_SHA_256)
2023-09-12 15:44:54,441 [env-execution-context-138] INFO  c.d.c.c.k.a.a.
↪AwsRequestResponseLogger:participant=participant1
↪tid:40d47592f1bd50f37e6804fbdff404dd - Sending request [e28450df3a98ea23]:
↪SignRequest(KeyId=f23b5b37-b4e8-494d-b2bc-1fca12308c99, Message=** Sign message
↪text placeholder **, MessageType=RAW, SigningAlgorithm=ECDSA_SHA_256) to https:/
↪/kms.us-east-1.amazonaws.com/
```

(continues on next page)

(continued from previous page)

```
2023-09-12 15:44:54,554 [aws-java-sdk-NettyEventLoop-1-2] INFO  c.d.c.c.k.a.a.
↪AwsRequestResponseLogger:participant=participant1
↪tid:40d47592f1bd50f37e6804fbdff404dd - Received response [e28450df3a98ea23]:
↪[Aws-Id: 7085bcf3-1a36-4048-a38b-014b441afa11] -
↪SignResponse(KeyId=arn:aws:kms:us-east-1:724647588434:key/f23b5b37-b4e8-494d-
↪b2bc-1fca12308c99, Signature=** Signature message text placeholder **,
↪SigningAlgorithm=ECDSA_SHA_256)
```

Note that sensitive data is removed before logging. The general log format is as follows:

```
tid:<canton_trace_id> - Sending request [<canton_kms_request_id>]:
<request details>                    tid:<canton_trace_id> - Received response
[<canton_kms_request_id>]: [Aws-Id: <aws_request_id>] - <response details>
```

1.31.4.3 Ledger-API Authorization

The Ledger API provides *authorization support* using JWT tokens. While the JWT token authorization allows third party applications to be authorized properly, it poses some issues for Canton internal services such as the *PingService* or the *DarService*, which are used to manage domain wide concerns. Therefore Canton generates a new admin bearer token (64 bytes, randomly generated, hex-encoded) on each startup, which is communicated to these services internally and used by these services to authorize themselves on the Ledger API. The admin token allows to act as any party registered on that participant node.

The admin token is only used within the same process. Therefore, in order to obtain this token, an attacker needs to be able to either dump the memory or capture the network traffic, which typically only a privileged user can do.

It is important to enable TLS together with JWT support in general, as otherwise tokens can be leaked to an attacker that has the ability to inspect network traffic.

1.32 Scaling and Performance

1.32.1 Network Scaling

The scaling and performance characteristics of a Canton-based system are determined by many factors. The simplest approach is to deploy Canton as a simple monolith where vertical scaling would add more CPUs, memory, etc. to the compute resource. However, the most frequent and expected deployment of Canton is as a distributed, micro-service architecture, running in different data centers of different organizations, with many opportunities to incrementally increase throughput. This is outlined below.

The ledger state in Canton does not exist globally so there is no single node that, by design, hosts all contracts. Instead, participant nodes are involved in transactions that operate on the ledger state on a strict need-to-know basis (data minimization), only exchanging (encrypted) information on the domains used as coordination points for the given input contracts. For example, if participants Alice and Bank transact on an i-owe-you contract on domain A, another participant Bob, or another domain B, does not receive a single bit related to this transaction. This is in contrast to blockchains, where each node has to process each block regardless of how active or directly affected they are by a given transaction. This lends itself to a micro-service approach that can scale horizontally.

The micro-services deployment of Canton includes the set of participant and domain nodes (hereafter, participant or participants and domain or domains respectively), as well as the services internal to the domain (e.g., Topology Manager). In general, each Canton micro-service follows the best practice of having its own local database which increases throughput. Deploying a service to its own compute server increases throughput because of the additional CPU and disk capacity. A vertical scaling approach can be used to increase throughput if a single service becomes a bottleneck, along with the option of horizontal scaling that is discussed next.

An initial Canton deployment can increase its scaling in multiple ways that build on each other. If a single participant node has many parties, then throughput can be increased by migrating parties off to a new, additional participant node (currently supported as a manual early access feature). For example, if 100 parties are performing multi-lateral transactions with each other, then the system can reallocate parties to 10 participants with 10 parties each, or 100 participants with 1 party each. As most of the computation occurs on the participants, a domain can sustain a very substantial load from multiple participants. If the domain were to be a bottleneck then the Sequencer(s), Topology Manager, and Mediator can be run on their own compute server which increases the domain throughput. Therefore, new compute servers with additional Canton nodes can be added to the network when needed, allowing the entire system to scale horizontally.

If even more throughput is needed then the multiple-domain feature of Canton can be leveraged to increase throughput. In a large and active network where a domain reaches the capacity limit, additional domains can be rolled out, such that the workflows can be sharded over the available domains (early access). This is a standard technique for load balancing where the client application does the load balancing via sharding.

If a single party is a bottleneck then the throughput can be increased by sharding the workflow across multiple parties hosted on separate participants. If a workflow is involving some large operator (i.e. an exchange), then an option would be to shard the operator by creating two operator parties and distribute the workflows evenly over the two operators (eventually hosted on different participants), and by adding some intermediate steps for the few cases where the workflows would span across the two shards.

Some anti-patterns need to be avoided for the maximum scaling opportunity. For example, having almost all of the parties on a single participant is an anti-pattern to be avoided since that participant will be a bottleneck. Similarly, the design of the Daml model has a strong impact on the degree to which sharding is possible. For example, having a Daml application that introduces a synchronization party through which all transactions need to be validated introduces a bottleneck so it is also an anti-pattern to avoid.

The bottom line is that a Canton system can scale out horizontally if commands involve only a small number of participants and domains.

Important: This feature is only available in Canton Enterprise

---

1.32.2 Node Scaling

The Daml Enterprise edition of Canton supports the following scaling of nodes:

The database-backed drivers (Postgres and Oracle) can run in an active-active setup with parallel processing, supporting multiple writer and reader processes. Thus, such nodes can scale horizontally.
The enterprise participant node processes transactions in parallel (except the process of conflict detection which by definition must be sequential), allowing much higher throughput than the community version. The community version is processing each transaction sequentially. Canton processes make use of multiple CPUs and will detect the number of available CPUs automatically. The number of parallel threads can be controlled by setting the JVM properties *scala.concurrent.context.numThreads* to the desired value.

Generally, the performance of Canton nodes is currently storage I/O bound. Therefore, their performance depends on the scaling behavior and throughput performance of the underlying storage layer, which can be a database or a distributed ledger for some drivers. Therefore, appropriately sizing the database is key to achieving the necessary performance.

On a related note: the Daml interpretation is a pure operation, without side-effects. Therefore, the interpretation of each transaction can run in parallel, and only the conflict detection between transactions must run sequentially.

1.32.3 Performance and Sizing

A Daml workflow can be computationally arbitrarily complex, performing lots of computation (cpu!) or fetching many contracts (io!), and involve different numbers of parties, participants, and domains. Canton nodes store their entire data in the storage layer (database), with additional indexes. Every workflow and topology is different, and therefore, sizing requirements depend on the Daml application that is going to run, and on the resource requirements of the storage layer. Therefore, to obtain sizing estimates you must measure the resource usage of dominant workflows using a representative topology and setup of your use case.

1.32.4 Batching

As every transaction comes with an overhead (signatures, symmetric encryption keys, serialization and wrapping into messages for transport, HTTP headers, etc), we recommend designing the applications submitting commands in a way that batches smaller requests together into a single transaction.

Optimal batch sizes depend on the workflow and the topology and need to be determined experimentally.

1.32.5 Asynchronous Submissions

In order to achieve best performance, we suggest that you use asynchronous command submissions. However, please note that the async submission is only partially asynchronous, as the initial command interpretation and transaction building is included in that step, while the transaction validation and result finalization is not. This means that an async submission takes between 50 to 1000 ms, depending on command size and complexity. In the extreme case with a single thread submitting transactions, this would mean that you would only achieve a rate of one command per second.

If you use synchronous command submissions, the system will wait for the entire transaction to complete, which will require even more threads. Also, please note that the synchronous command submission has a default upper limit of 256 in flight commands, which can be reconfigured using

```
canton.participants.participant1.ledger-api.command-service.max-commands-in-
↪flight = 256 // default value
```

1.32.6 Storage Estimation

A priori storage estimation of a Canton installation is tricky. As explained above, storage usage depends on topology, payload, Daml models used, and what type of storage layer is configured. However, the following example may help you understand the storage usage for your use case:

First, a command submitted through the ledger API is sent to the participant as a serialized gRPC request.

This command is first interpreted and translated into a Daml-LF transaction. The interpreted transaction is next translated into a Canton transaction view-decomposition, which is a privacy-preserving representation of the full transaction tree structure. A transaction typically consists of several transaction views; in the worst case, every action node in the transaction tree becomes a separate transaction view. Each view contains the full set of arguments required by that view, including the contract arguments of the input contracts. So the data representation can be multiplied quite a bit. Here, we cannot estimate the resulting size without having a concrete example. For simplicity, let us consider the simple case where a participant is exercising a simple Transfer choice on an typical Iou contract to a new owner, preserving the other contract arguments. We assume that the old and new owners of the IOU are hosted on the same participant whereas the IOU issuer is hosted on a second participant.

The resulting Canton transaction consists of two views (one for the **Exercise** node of the Transfer choice and one for the **Create** node of the transferred IOU). Both views contain some metadata such as the package and template identifiers, contract keys, stakeholders, and involved participants. The view for the **Exercise** node contains the contract arguments of the input IOU, say of size *Y*. The view for the **Create** node contains the updated contract arguments for the created contract, again of size *Y*. Note that there is no fixed relation between the command size *X* and the size of the input contracts *Y*. Typically *X* only contains the receiver of the transfer, but not the contract arguments that are stored on the ledger.

Then, we observe the following storage usage:

> Two encrypted envelopes with payload *Y* each, one symmetric key per view and informee participant of that view, two root hashes for each participant and the participant IDs as recipients at the sequencer store, and the informee tree for the mediator (informees and transaction metadata, but no payload), together with the sequencer database indexes.

Two encrypted envelopes with payload *Y* each and the symmetric keys for the views, in the participant events table of each participant (as both receive the data)
Decrypted new resulting contract of size *Y* in the private contract store and some status information of that contract on the active contract journal of the sync service.
The full decrypted transaction with a payload of size *Y* for the created contract, in the sync service linear event log. This transaction does not contain the input contract arguments.
The full decrypted transaction with *Y* in the indexer events table, excluding input contracts, but including newly divulged input contracts.

If we assume that payloads dominate the storage requirements, we conclude that the storage requirement is given by the payload multiplication due to the view decomposition. In our example, the transaction requires *5*Y* storage on each participant and *2*Y* on the sequencer. For the two participants and the sequencer, this makes *12*Y* in total.

Additionally to this, some indexes have to be built by the database to serve the contracts and events efficiently. The exact estimation of the size usage of such indexes for each database layer is beyond the scope of our documentation.

Note: Please note that we do have plans to remove the storage duplication between the sync service and the indexer. Ideally, will be able to reduce the storage on the participant for this example from *5*Y* down to *3*Y*: once for the unencrypted created contract and twice for the two encrypted transaction views.

Generally, to recover used storage, a participant and a domain can be pruned. Pruning is available on Canton Enterprise through a *set of console commands* and allows removal of past events and archived contracts based on a timestamp. The storage usage of a Canton deployment can be kept constant by continuously removing obsolete data. Non-repudiation and auditability of the unpruned history are preserved due to the bilateral commitments.

1.32.7 Set Up Canton to Get the Best Performance

In this section, the findings from internal performance tests are outlined to help you achieve optimal performance for your Canton application.

1.32.7.1 System Design / Architecture

We recommend the version of Canton included in the Daml Enterprise edition, which is heavily optimized when compared with the community edition.

Plan your topology such that your Daml parties can be partitioned into independent blocks. That means most of your Daml commands involve parties of a single block only. It is ok if some commands involve parties of several (or all) blocks, as long as this happens only very rarely. In particular, avoid having a single master party that is involved in every command, because that party bottlenecks the system.

If your participants are becoming a bottleneck, add more participant nodes to your system. Make sure that each block runs on its own participant. If your domain(s) are becoming a bottleneck, add more domain nodes and distribute the load evenly over all domains.

Prefer sending big commands with multiple actions (creates / exercises) over sending numerous small commands. Avoid sending unnecessary commands through the ledger API. Try to minimize

the payload of commands.

Further information can be found in Section *Scaling and Performance*.

1.32.7.2 Hardware and Database

Do not run Canton nodes with an in-memory storage or with an H2 storage in production or during performance tests. You may observe very good performance in the beginning, but performance can degrade substantially once the data stores fill up.

Measure memory usage, CPU usage and disk throughput and improve your hardware as needed. For simplicity, it makes sense to start on a single machine. Once the resources of a machine are becoming a bottleneck, distribute your nodes and databases to different machines.

Try to make sure that the latency between a Canton node and its database is very low (ideally in the order of microseconds). The latency between Canton nodes has a much lower impact on throughput than the latency between a Canton node and its database.

Optimize the configuration of your database, and make sure the database has sufficient memory and is stored on SSD disks with a very high throughput. For Postgres, this online tool is a good starting point for finding reasonable parameters.

1.32.7.3 Configuration

In the following, we go through the parameters with known impact on performance.

Timeouts. Under high load, you may observe that commands timeout. This will negatively impact throughput, because the commands consume resources without contributing to the number of accepted commands. To avoid this situation increase timeout parameters from the Canton console:

```
myDomain.service.update_dynamic_domain_parameters(
  _.update(
    participantResponseTimeout = 60.seconds,
    mediatorReactionTimeout = 60.seconds
  )
)
```

If timeouts keep occurring, change your setup to submit commands at a lower rate. In addition, take the next paragraph on resource limits into account.

Tune resource limits. Resource limits are used to prevent ledger applications from overloading Canton by sending commands at an excessive rate. While resource limits are necessary to protect the system from denial of service attacks in a production environment, they can prevent Canton from achieving maximum throughput. Resource limits can be configured as follows from the Canton console:

```
participant1.resources.set_resource_limits(
  ResourceLimits(
    // Allow for submitting at most 200 commands per second
    maxRate = Some(200),

    // Limit the number of in-flight requests to 500.
    // A "request" includes every transaction that needs to be validated by
↪participant1:
```

(continues on next page)

(continued from previous page)

```
    // - transactions originating from commands submitted to participant1
    // - transaction originating from commands submitted to different
↪participants.
    // The chosen configuration allows for processing up to 100 requests per
↪second
    // with an average latency of 5 seconds.
    maxDirtyRequests = Some(500),

    // Allow submission bursts of up to `factor * maxRate`
    maxBurstFactor = 0.5,
  )
)
```

As a rule of thumb, configure `maxDirtyRequests` to be slightly larger than `throughput * latency`, where

> `throughput` is the number of requests per second Canton needs to handle and
> `latency` is the time taken to process a single request while Canton is receiving requests at rate `throughput`.

You should run performance tests to ensure that `throughput` and `latency` are actually realistic. Otherwise, an application may overload Canton by submitting more requests than Canton can handle.

Configure the `maxRate` parameter to be slightly higher than the expected maximal `throughput`.

If you need to support command bursts, configure the `maxBurstFactor` accordingly. Then, the `maxRate` limitation will only start to enforce the rate after having received the initial burst of `maxBurstFactor * maxRate`.

To find optimal resource limits you need to run performance tests. The `maxDirtyRequest` parameter will protect Canton from being overloaded, if requests are arriving at a constant rate. The `maxRate` parameter offers additional protection, if requests are arriving at a variable rate.

If you choose higher resource limits, you may observe a higher throughput, at the risk of a higher latency. In the extreme case however, latency grows so much that commands will timeout; as a result, the command processing consumes resources even though some commands are not committed to the ledger.

If you choose lower resource limits, you may observe a lower latency, at the cost of lower throughput and commands getting rejected with the error code `PARTICIPANT_BACKPRESSURE`.

Size of connection pools. Make sure that every node uses a connection pool to communicate with the database. This avoids the extra cost of creating a new connection on every database query. Canton chooses a suitable connection pool by default. Configure the maximum number of connections such that the database is fully loaded, but not overloaded. Allocating too many database connections will lead to resource waste (each thread costs), context switching and contention on the database system, slowing the overall system down. You can notice this on the query latencies reported by canton going up.

Try to observe the `db-storage.queue` metrics. If they are large, then the system performance may benefit from tuning the number of database connections. Detailed instructions can be found in the Section *Max Connection Settings*.

Throttling configuration for SequencerClient. The `SequencerClient` is the component responsible for managing the connection of any member (participant, mediator, or topology manager) in a

Canton network to the domain. Each domain can have multiple sequencers, and the `SequencerClient` connects to one of them. However, there is a possibility that the `SequencerClient` can become overwhelmed and struggle to keep up with the incoming messages. To address this issue, a configuration parameter called `maximum-in-flight-event-batches` is available:

```
domain-managers {
  domainManager1 {
    sequencer-client.maximum-in-flight-event-batches = 100
  }
}
participants {
  participant1 {
    sequencer-client.maximum-in-flight-event-batches = 100
  }
  participant2 {
    sequencer-client.maximum-in-flight-event-batches = 100
  }
}
mediators {
  mediator1 {
    sequencer-client.maximum-in-flight-event-batches = 100
  }
}
sequencers {
  sequencer1 {
    sequencer-client.maximum-in-flight-event-batches = 100
  }
}
```

By setting the `maximum-in-flight-event-batches` parameter, you can control the maximum number of event batches that the system processes concurrently. This configuration helps prevent overload and ensures that the system can handle the workload effectively.

It's important to note that the value you choose for `maximum-in-flight-event-batches` impacts the `SequencerClient`'s performance in several ways. A higher value can potentially increase the `SequencerClient`'s throughput, allowing it to handle more events simultaneously. However, this comes at the cost of higher memory consumption and longer processing times for each batch.

On the other hand, a lower value for `maximum-in-flight-event-batches` might limit the throughput, as it can process fewer events concurrently. However, this approach can result in more stable and predictable `SequencerClient` behavior.

To monitor the performance of the `SequencerClient` and ensure it is operating within the desired limits, you can observe the metric `sequencer-client.handler.actual-in-flight-event-batches`. This metric provides the current value of the in-flight event batches, indicating how close it is to the configured limit. Additionally, you can also reference the metric `sequencer-client.handler.max-in-flight-event-batches` to determine the configured maximum value.

By monitoring these metrics, you can gain insights into the actual workload being processed and assess whether it is approaching the specified limit. This information is valuable for maintaining optimal `SequencerClient` performance and preventing any potential bottlenecks or overload situations.

Size of database task queue. If you are seeing frequent `RejectedExecutionExceptions` when Canton queries the database, increase the size of the task queue, as described in Section *Database task queue full*. The rejection is otherwise harmless. It just points out that the database is overloaded.

Database Latency. Ensure that the database latency is low. The higher the database latency, the lower the actual bandwidth and the lower the throughput of the system.

Turn on High-Throughput Sequencer. The database sequencer has a number of parameters that can be tuned. The trade-off is low-latency or high-throughput. In the low-latency setting, every submission will be immediately processed as a single item. In the high-throughput setting, the sequencer will accumulate a few events before writing them together at once. While the latency added is only a few ms, it does make a difference during development and testing of your Daml applications. Therefore, the default setting is `low-latency`. A production deployment with high throughput demand should choose the `high-throughput` setting by configuring:

```
// example setting for domain nodes. database sequencer nodes have the exact same
↪settings.
canton.domains.mydomain.sequencer {
    type = database
    writer = {
        // choose between high-throughput or low-latency
        type = high-throughput
    }
}
```

There are additional parameters that can in theory be fine-tuned, but we recommend to leave the defaults and use either high-throughput or low-latency. In our experience, a high-throughput sequencer can handle several thousand submissions per second.

JVM heap size. In case you observe `OutOfMemoryErrors` or high overhead of garbage collection, you must increase the heap size of the JVM, as described in Section *Java Virtual Machine Arguments*. Use tools of your JVM provider (such as VisualVM) to monitor the garbage collector to check whether the heap size is tight.

Size of thread pools. Every Canton process has a thread pool for executing internal tasks. By default, the size of the thread-pool is configured as the number of (virtual) cores of the underlying (physical) machine. If the underlying machine runs other processes (e.g., a database) or if Canton runs inside of a container, the thread-pool may be too big, resulting in excessive context switching. To avoid that, configure the size of the thread pool explicitly like this:

```
"bin/canton -Dscala.concurrent.context.numThreads=12 --config examples/01-simple-
↪topology/simple-topology.conf"
```

As a result, Canton will log the following line:

```
"INFO  c.d.c.e.EnterpriseEnvironment - Deriving 12 as number of threads from '-
↪Dscala.concurrent.context.numThreads'."
```

Asynchronous commits. If you are using a Postgres database, configure the participant's ledger API server to commit database transactions asynchronously by including the following line into your Canton configuration:

```
canton.participants.participant1.ledger-api.postgres-data-source.synchronous-
↪commit = off
```

Logging Settings. Make sure that Canton outputs log messages only at level INFO and above and turn off immediate log flushing using the `--log-immediate-flush=false` commandline flag, at the risk of missing log entries during a host system crash.

Replication. If (and **only if**) using single nodes for participant, sequencer, and/or mediator, replica-

tion can be turned off by setting `replication.enabled = false` in their respective configuration.

Warning: While replication can be turned off to try to obtain performance gains, it must **not** be disabled when running multiple nodes for HA.

Caching Configuration. In some cases, you might also want to tune caching configurations and either reduce or increase them, depending on your situation. This can also be helpful if you need to reduce the memory foot-print of Canton, which can be large, as the default cache configurations are tailored for high-throughput, high-memory and small transaction sizes.

Generally, the caches that usually matter with respect to size are the contract caches and the in-memory fan-out event buffer. You can tune these using the following configurations. The values depicted here are the ones recommended for smaller memory-footprints and are therefore also helpful if you run into out-of-memory issues:

```
canton.participants.participant1 {
    // tune caching configs of the ledger api server
    ledger-api {
        max-contract-state-cache-size = 1000 // default 1e6
        max-contract-key-state-cache-size = 1000 // default 1e6

        // The in-memory fan-out will serve the transaction streams from memory
↪as they are finalized, rather than
        // using the database. Therefore, you should choose this buffer to be
↪large enough such that the likeliness of
        // applications having to stream transactions from the database is low.
↪Generally, having a 10s buffer is
        // sensible. Therefore, if you expect e.g. a throughput of 20 tx/s, then
↪setting this number to 200 is sensible.
        // The default setting assumes 1000 tx/s.
        max-transactions-in-memory-fan-out-buffer-size = 200 // default 1e5
    }
    // tune the synchronisation protocols contract store cache
    caching {
        contract-store {
            maximum-size = 1000 // default 1e6
            expire-after-access = 120s // default 10 minutes
        }
    }
}
```

1.33 Advanced Ledger Operations

1.33.1 Manage Domains

1.33.1.1 Permissioned Domains

Important: This feature is only available in Canton Enterprise

Canton as a network is an open virtual shared ledger. Whoever runs a Canton participant node is part of the same virtual shared ledger. However, the network itself is made up of domains that are used by participants to run the Canton protocol and communicate to their peers. Such domains can be *open*, allowing any participant with access to a sequencer node to enter and participate in the network. But domains can also be `permissioned`, where the operator of the domain topology managers needs to explicitly add the participant to the allow-list before the participant can register with a domain.

While the Canton architecture is designed to be resilient against malicious participants, there can never be a guarantee that the implementation of said architecture is absolutely secure. Therefore, it makes sense for most networks to impose control on which participant can be part of the network.

The first layer of control is given by securing access to the public api of the sequencers in the network. This can be done using standard network tools such as firewalls and virtual private networks.

The second layer of control is given by setting the appropriate configuration flag of the domain manager (or domain):

```
canton.domain-managers.domainManager1.topology.open = false
```

Assuming we have set up a domain with this flag turned off, the config for that particular domain would read:

```
@ val config = DomainConnectionConfig("mydomain", sequencer1.sequencerConnection)
config : DomainConnectionConfig = DomainConnectionConfig(
  domain = Domain 'mydomain',
  sequencerConnections = Sequencer 'DefaultSequencer' -> GrpcSequencerConnection(
    endpoints = http://127.0.0.1:30078,
    transportSecurity = false,
..
```

When a participant attempts to join the domain, it will be rejected:

```
@ participant1.domains.register(config)
ERROR com.digitalasset.canton.integration.EnterpriseEnvironmentDefinition$$anon$3
 ↪- Request failed for participant1.
  GrpcRequestRefusedByServer: FAILED_PRECONDITION/PARTICIPANT_IS_NOT_ACTIVE(9,
 ↪20d5cc73): The participant is not yet active
  Request: RegisterDomain(DomainConnectionConfig(
  domain = Domain 'mydomain',
  sequencerConnections = Sequencer 'DefaultSequencer' ->
 ↪GrpcSequencerConnection(endpoints = http://127.0.0.1:30078, transportSecurity =
 ↪false, customTrustCertificates = None()),
    ...
  CorrelationId: 20d5cc7385cb9dc0895e47ef8fc0495f
  Context: HashMap(participant -> participant1, test ->
 ↪ManagePermissionedDomainsDocumentationManual, serverResponse -> Domain Domain
 ↪'mydomain' has rejected our on-boarding attempt, domain -> mydomain, tid ->
 ↪20d5cc7385cb9dc0895e47ef8fc0495f)
  Command ParticipantAdministration$domains$.register invoked from cmd10000006.
 ↪sc:1
```

In order to allow the participant to join the domain, we must first actively enable it on the topology manager. We assume now that the operator of the participant *extracts its id* into a string:

```
@ val participantAsString = participant1.id.toProtoPrimitive
participantAsString : String =
↪"PAR::participant1::12201c55b3f9bd2090569099ed1a2eb60a872cad7170cb286a5e09b01bf32d856037
↪"
```

and communicates this string to the operator of the domain topology manager:

```
@ val participantIdFromString = ParticipantId.
↪tryFromProtoPrimitive(participantAsString)
participantIdFromString : ParticipantId = PAR::participant1::12201c55b3f9...
```

This topology manager can now add the participant by enabling it:

```
@ domainManager1.participants.set_state(participantIdFromString,
↪ParticipantPermission.Submission, TrustLevel.Ordinary)
```

Note that the participant is not active yet:

```
@ domainManager1.participants.active(participantIdFromString)
res5: Boolean = false
```

So far, what we've done with setting the state is to issue a domain trust certificate , where the domain topology manager declares that it trusts the participant enough to become a participant of the domain. We can inspect this certificate using:

```
@ domainManager1.topology.participant_domain_states.list(filterStore="Authorized
↪").map(_.item)
res6: Seq[ParticipantState] = Vector(
  ParticipantState(
    From,
    domainManager1::1220ce9a486a...,
    PAR::participant1::12201c55b3f9...,
    Submission,
    Ordinary
  )
)
```

In order to have the participant become active on the domain, we need to register the signing keys and the domain trust certificate of the participant. The certificate is generated by the participant automatically and sent to the domain during the initial handshake.

We can trigger that handshake again by attempting to reconnect to the domain again:

```
@ participant1.domains.reconnect_all()
```

Now, we can check that the participant is active:

```
@ domainManager1.participants.active(participantIdFromString)
res8: Boolean = true
```

We can also observe that we now have both sides of the domain trust certificate, the `From` and the `To`:

```
@ domainManager1.topology.participant_domain_states.list(filterStore="Authorized
↪").map(_.item)
res9: Seq[ParticipantState] = Vector(
```

(continues on next page)

(continued from previous page)

```
  ParticipantState(
    From,
    domainManager1::1220ce9a486a...,
    PAR::participant1::12201c55b3f9...,
    Submission,
    Ordinary
  ),
  ParticipantState(
    To,
    domainManager1::1220ce9a486a...,
    PAR::participant1::12201c55b3f9...,
    Submission,
    Ordinary
  )
)
```

Finally, the participant is healthy and can use the domain:

```
@ participant1.health.ping(participant1)
res10: Duration = 2646 milliseconds
```

1.33.1.2 Domain Rules

Every domain has its own rules in terms of what parameters are used by the participants while running the protocol. The participants obtain these parameters before connecting to the domain. They can be configured using the specific parameter section. An example would be:

```
init.domain-parameters {
  // example setting
  unique-contract-keys = yes
}
```

The full set of available parameters can be found in the scala reference documentation.

1.33.1.3 Dynamic domain parameters

In addition to the parameters that are specified in the configuration, some parameters can be changed at runtime (i.e., while the domain is running); these are called **dynamic domain parameters**. When the domain is bootstrapped, default values are used for the dynamic domain parameters. They can be changed subsequently using the console commands described below.

A participant can get the current parameters on a domain it is connected to using the following command:

```
mydomain.service.get_dynamic_domain_parameters
```

Parameters that were transitioned from static to dynamic with protocol version 4 need to be retrieved individually:

```
mydomain.service.get_reconciliation_interval
mydomain.service.get_max_rate_per_participant
```

(continues on next page)

(continued from previous page)

```
mydomain.service.get_max_request_size
mydomain.service.get_mediator_deduplication_timeout
```

Dynamic parameters can bet set individually using:

```
mydomain.service.set_reconciliation_interval(5.seconds)
mydomain.service.set_max_rate_per_participant(100)
mydomain.service.set_max_request_size(100000)
mydomain.service.set_mediator_deduplication_timeout(2.minutes)
```

Alternatively, several can be set at the same time:

```
mydomain.service.update_dynamic_domain_parameters(
  _.update(
    participantResponseTimeout = 10.seconds,
    topologyChangeDelay = 1.second,
  )
)
```

Note: When increasing *max request size*, the sequencer nodes need to be restarted for the new value to be taken into account. If the domain is not distributed, it means that the domain node needs to be restarted.

1.33.1.4 Recover From a Small Max Request Size

MaxRequestSize is a dynamic parameter starting from protocol version 4. This parameter configures both the grpc channel size on the sequencer node and the maximum size that a sequencer client is allowed to transfer.

If the parameter is set to a very small value (roughly under *30kb*), Canton can crash because all messages are rejected by the sequencer client or by the sequencer node. This cannot be corrected by setting a higher value within the console, because this change request needs to be send via the sequencer and will also be rejected.

To recover from this crash, you need to configure *override-max-request-size* on both the sequencer node and the sequencer clients.

On a non-distributed deployment, this means modifying both the domain and the participants configuration as follows:

```
domains {
  da {
    # overrides the maxRequestSize in bytes on the sequencer node
    public-api.override-max-request-size = 30000
    sequencer-client.override-max-request-size = 30000
  }
}
participants {
  participant1 {
    sequencer-client.override-max-request-size = 30000
  }
```

(continues on next page)

(continued from previous page)

```
  participant2 {
    sequencer-client.override-max-request-size = 30000
  }
}
```

On a distributed deployment, for each domain entity deployed on its own node, you will need to override the *max-request-size* as follows:

```
domain-managers {
  domainManager1 {
    sequencer-client.override-max-request-size = 30000
  }
}
participants {
  participant1 {
    sequencer-client.override-max-request-size = 30000
  }
  participant2 {
    sequencer-client.override-max-request-size = 30000
  }
}
mediators {
  mediator1 {
    sequencer-client.override-max-request-size = 30000
  }
}
sequencers {
  sequencer1 {
    # overrides the maxRequestSize in bytes on the sequencer node
    public-api.override-max-request-size = 30000
    sequencer-client.override-max-request-size = 30000
  }
}
```

After the configuration is modified, disconnect all the participants from the domain and then restart all nodes.

On a non-distributed deployment, you can stop Canton by following these steps:

```
participants.all.domains.disconnect(da.name)
nodes.local.stop()
```

On a distributed deployment, you can stop Canton by following these steps:

```
participants.all.domains.disconnect(sequencer1.name)
nodes.local.stop()
```

Then perform the restart:

```
nodes.local.start()
participants.all.domains.reconnect_all()
```

Once Canton has recovered, use the admin command to set the *maxRequestSize* value, then delete the added configuration in the previous step, and finally perform the restart again.

1.33.2 Manage Domain Entities

1.33.2.1 Setting up a Distributed Domain With a Single Console

If you're running a domain node in its default configuration as shown previously in this current page, it will have a sequence and mediator embedded and these components will be automatically bootstrapped for you.

If your domain operates with external sequencers and mediators, you will need to configure a domain manager node (which only runs topology management) and bootstrap your domain with at least one external sequencer node and one external mediator node.

First make sure the nodes are fresh and have not yet been initialized:

```
@ mediator1.health.initialized()
res1: Boolean = false
```

```
@ sequencer1.health.initialized()
res2: Boolean = false
```

```
@ domainManager1.health.initialized()
res3: Boolean = false
```

The domain manager also needs its identity to be generated and ready before we can bootstrap the domain:

```
@ domainManager1.health.wait_for_identity()
```

Note: This is technically only required when accessing the domain manager through a remote console, but is a good practice regardless.

Now you can initialize the distributed domain as follows:

```
@ domainManager1.setup.bootstrap_domain(Seq(sequencer1), Seq(mediator1))
```

At this point a participant should be able to connect to a sequencer and operate on that domain:

```
@ participant1.domains.connect_local(sequencer1)
```

```
@ participant1.health.ping(participant1)
res7: Duration = 5514 milliseconds
```

Domain managers are configured as `domain-managers` under the `canton` configuration. Domain managers are configured similarly to domain nodes, except that there are no sequencer, mediator, public api or service agreement configs.

Please note that if your sequencer is database-based and you're horizontally scaling it as described under *sequencer high availability*, you do not need to pass all sequencer nodes into the command above. Since they all share the same relational database, you only need to run this initialization step on one of them.

For non-database-based sequencer such as Ethereum or Fabric sequencers you need to have each node initialized individually. You can either initialize such sequencers as part of the initial domain

bootstrap shown above or dynamically add a new sequencer at a later point as described in *operational processes*.

1.33.2.2 Setting up a Distributed Domain With Separate Consoles

The process outlined in the previous section only works if all nodes are accessible from the same console environment. If each node has its own isolated console environment, the bootstrapping process must be coordinated in steps with the exchange of data via files using any secure channel of communication between the environments.

Note: Please ensure that all of the nodes in the distributed domain are started before proceeding.

Initially the domain manager must transmit its domain parameters from its console by saving the parameters to a file. The domain id, serialized as a string, must also be transmitted.

```
@ domainManager1.service.get_static_domain_parameters.writeToFile("tmp/domain-
↪bootstrapping-files/params.proto")
```

```
@ val domainIdString = domainManager1.id.toProtoPrimitive
domainIdString : String =
↪"domainManager1::1220be641b8b69c9e56c6548ac78437f05e2fdc0be96df9a70dfe2e403d28da0de9b
↪"
```

Then the sequencer must receive this file, deserialize it and initialize itself. As part of the initialization, the sequencer creates a signing key pair whose public key it must then transmit via file. Optionally, repeat this for any extra sequencer nodes.

```
@ val domainParameters = com.digitalasset.canton.admin.api.client.data.
↪StaticDomainParameters.tryReadFromFile("tmp/domain-bootstrapping-files/params.
↪proto")
domainParameters : StaticDomainParameters = StaticDomainParametersV1(
  uniqueContractKeys = true,
  requiredSigningKeySchemes = Set(Ed25519, ECDSA-P256, ECDSA-P384),
  requiredEncryptionKeySchemes = Set(ECIES-P256_HMAC256_AES128-GCM),
  requiredSymmetricKeySchemes = Set(AES128-GCM),
  requiredHashAlgorithms = Set(Sha256),
  requiredCryptoKeyFormats = Set(Tink),
  protocolVersion = 4
)
```

```
@ val domainId = DomainId.tryFromString(domainIdString)
domainId : DomainId = domainManager1::1220be641b8b...
```

```
@ val initResponse = sequencer1.initialization.initialize_from_beginning(domainId,
↪ domainParameters)
initResponse : com.digitalasset.canton.domain.sequencing.admin.grpc.
↪InitializeSequencerResponse = InitializeSequencerResponse(
  keyId = "sequencer-id",
  publicKey = SigningPublicKey(id = 122050ae909a..., format = Tink, scheme =
↪Ed25519),
  replicated = false
)
```

```
@ initResponse.publicKey.writeToFile("tmp/domain-bootstrapping-files/seq1-key.
↪proto")
```

The domain manager must then authorize the sequencer's key. Optionally, repeat this for any extra sequencer keys.

```
@ val sequencerPublicKey = SigningPublicKey.tryReadFromFile("tmp/domain-
↪bootstrapping-files/seq1-key.proto")
sequencerPublicKey : SigningPublicKey = SigningPublicKey(id = 122050ae909a...,
↪format = Tink, scheme = Ed25519)
```

```
@ domainManager1.setup.helper.authorizeKey(sequencerPublicKey, "sequencer",
↪SequencerId(domainManager1.id))
```

Now the mediator also needs to create a signing key pair and transmit it. Optionally, repeat this for any extra mediator nodes.

```
@ mediator1.keys.secret.generate_signing_key("initial-key").writeToFile("tmp/
↪domain-bootstrapping-files/med1-key.proto")
```

The domain manager must now authorize the mediator's key and also authorize the mediator to act as part of this domain. Optionally, repeat this for any extra mediator nodes.

```
@ val mediatorKey = SigningPublicKey.tryReadFromFile("tmp/domain-bootstrapping-
↪files/med1-key.proto")
mediatorKey : SigningPublicKey = SigningPublicKey(id = 122097079077..., format =
↪Tink, scheme = Ed25519)
```

```
@ val domainId = DomainId.tryFromString(domainIdString)
domainId : DomainId = domainManager1::1220be641b8b...
```

```
@ domainManager1.setup.helper.authorizeKey(mediatorKey, "mediator1",
↪MediatorId(domainId))
```

```
@ domainManager1.topology.mediator_domain_states.authorize(TopologyChangeOp.Add,
↪domainId, MediatorId(domainId), RequestSide.Both)
res13: com.google.protobuf.ByteString = <ByteString@227271fd size=560 contents="\
↪n\255\004\n\333\001\n\326\001\n\323\001\022 TJu4dWv2cpqPUOi2StZtQyuEE2BjPM0UR...
↪">
```

After that, still on the domain manager's console, the domain manager must collect the list of topology transactions, which include all the key authorizations and a few other things it needs to broadcast to all domain members. This is now saved to a file.

```
@ domainManager1.topology.all.list().collectOfType[TopologyChangeOp.Positive].
↪writeToFile("tmp/domain-bootstrapping-files/topology-transactions.proto")
```

The sequencer then reads this set of initial topology transactions and sequences it as the first message to be sequenced in this domain. This will allow the domain members whose keys were authorized in previous steps to connect to this sequencer and operate with it. The sequencer will then transmit its connection info.

```
@ val initialTopology = com.digitalasset.canton.topology.store.
↪StoredTopologyTransactions.tryReadFromFile("tmp/domain-bootstrapping-files/
↪topology-transactions.proto").collectOfType[TopologyChangeOp.Positive]
initialTopology : store.StoredTopologyTransactions[TopologyChangeOp.Positive] =
↪Seq(
  StoredTopologyTransaction(
    sequenced = 2023-06-12T12:19:31.150925Z,
    validFrom = 2023-06-12T12:19:31.150925Z,
    validUntil = 2023-06-12T12:19:31.150925Z,
    op = Add,
..
```

```
@ sequencer1.initialization.bootstrap_topology(initialTopology)
```

```
@ SequencerConnections.single(sequencer1.sequencerConnection).writeToFile("tmp/
↪domain-bootstrapping-files/sequencer-connection.proto")
```

To initialize the mediator, it will need a connection to the sequencer and the domain parameters. Optionally, repeat this for any extra mediator nodes.

```
@ val sequencerConnections = SequencerConnections.tryReadFromFile("tmp/domain-
↪bootstrapping-files/sequencer-connection.proto")
sequencerConnections : SequencerConnections = Sequencer 'DefaultSequencer' ->
↪GrpcSequencerConnection(
  endpoints = http://127.0.0.1:30141,
  transportSecurity = false,
  customTrustCertificates = None()
)
```

```
@ val domainParameters = com.digitalasset.canton.admin.api.client.data.
↪StaticDomainParameters.tryReadFromFile("tmp/domain-bootstrapping-files/params.
↪proto")
domainParameters : StaticDomainParameters = StaticDomainParametersV1(
  uniqueContractKeys = true,
  requiredSigningKeySchemes = Set(Ed25519, ECDSA-P256, ECDSA-P384),
  requiredEncryptionKeySchemes = Set(ECIES-P256_HMAC256_AES128-GCM),
  requiredSymmetricKeySchemes = Set(AES128-GCM),
  requiredHashAlgorithms = Set(Sha256),
  requiredCryptoKeyFormats = Set(Tink),
  protocolVersion = 4
)
```

```
@ mediator1.mediator.initialize(domainId, MediatorId(domainId), domainParameters,
↪sequencerConnections, None)
res20: PublicKey = SigningPublicKey(id = 1220189c84ae..., format = Tink, scheme =
↪Ed25519)
```

```
@ mediator1.health.wait_for_initialized()
```

The domain manager will also need a connection to the sequencer in order to complete its initialization .

```
@ val sequencerConnection = SequencerConnections.tryReadFromFile("tmp/domain-
↪bootstrapping-files/sequencer-connection.proto")
```

(continues on next page)

(continued from previous page)

```
sequencerConnection : SequencerConnections = Sequencer 'DefaultSequencer' ->
↪GrpcSequencerConnection(
  endpoints = http://127.0.0.1:30141,
  transportSecurity = false,
  customTrustCertificates = None()
)
```

```
@ domainManager1.setup.init(sequencerConnection)
```

```
@ domainManager1.health.wait_for_initialized()
```

At this point the distributed domain should be completely initialized and a participant should be able to operate on this domain by connection to the sequencer.

```
@ participant1.domains.connect_local(sequencer1)
```

```
@ participant1.health.ping(participant1)
res26: Duration = 948 milliseconds
```

Additionally, please note that if more than one sequencers have been initialized, any mediator node and domain manager can choose to connect to just a subset of them.

1.33.2.3 Adding new sequencers to distributed domain

For non-database-based sequencers such as Ethereum or Fabric sequencers, you can either initialize them as part of the regular *distributed domain bootstrapping process* or dynamically add a new sequencer at a later point as follows:

```
domainManager1.setup.onboard_new_sequencer(
  initialSequencer = sequencer1,
  newSequencer = sequencer2,
)
```

Similarly to *initializing a distributed domain with separate consoles*, dynamically onboarding new sequencers (supported by Fabric and Ethereum sequencers) can be achieved in separate consoles as follows:

```
// Second sequencer's console: write signing key to file
{
  secondSequencer.keys.secret
    .generate_signing_key(s"${secondSequencer.name}-signing")
    .writeToFile(file1)
}

// Domain manager's console: write domain params and current topology
{
  domainManager1.service.get_static_domain_parameters.writeToFile(paramsFile)

  val sequencerSigningKey = SigningPublicKey.tryReadFromFile(file1)

  domainManager1.setup.helper.authorizeKey(
    sequencerSigningKey,
```

(continues on next page)

(continued from previous page)

```
    s"${secondSequencer.name}-signing",
    sequencerId,
  )

  domainManager1.setup.helper.waitForKeyAuthorizationToBeSequenced(
    sequencerId,
    sequencerSigningKey,
  )

  domainManager1.topology.all
    .list(domainId.filterString)
    .collectOfType[TopologyChangeOp.Positive]
    .writeToFile(file1)
}

// Initial sequencer's console: read topology and write snapshot to file
{
  val topologySnapshotPositive =
    StoredTopologyTransactions
      .tryReadFromFile(file1)
      .collectOfType[TopologyChangeOp.Positive]

  val sequencingTimestamp = topologySnapshotPositive.lastChangeTimestamp.
↪getOrElse(
    sys.error("topology snapshot is empty")
  )

  sequencer.sequencer.snapshot(sequencingTimestamp).writeToFile(file2)
}

// Second sequencer's console: read topology, snapshot and domain params
{
  val topologySnapshotPositive =
    StoredTopologyTransactions
      .tryReadFromFile(file1)
      .collectOfType[TopologyChangeOp.Positive]

  val state = SequencerSnapshot.tryReadFromFile(file2)

  val domainParameters = StaticDomainParameters.tryReadFromFile(paramsFile)

  secondSequencer.initialization
    .initialize_from_snapshot(
      domainId,
      topologySnapshotPositive,
      state,
      domainParameters,
    )
    .publicKey

  secondSequencer.health.initialized() shouldBe true

}
```

1.33.3 Ledger Pruning

Pruning refers to the selective removal of old, stale, or unneeded data from participant, domain sequencer, and mediator nodes. Nodes operate continuously for an indefinite amount of time on a limited amount of storage. In addition, privacy demands may require removing Personally Identifiable Information (PII) upon request.

Pruning participant nodes means removing archived contracts (and associated transactions and events). Pruning never removes active (i.e., non-archived) Daml contracts. For domain sequencers and mediators, pruning relates to the removal of processed messages. Participants and domain sequencers and mediators can have different pruning schedules set.

1.33.3.1 Enable Automatic Pruning

Enable automatic pruning by specifying a pruning schedule consisting of the following:

A cron expression that designates regular pruning begin times .
A maximum duration specifying pruning end times relative to the begin times of the cron expression.
A retention period to specify how far to prune relative to the current time.

For example, to run pruning every Saturday starting at 8am until 4pm (both in UTC):

```
participant.pruning.set_schedule("0 0 8 ? * SAT", 8.hours, 90.days)
domain.sequencer.pruning.set_schedule("0 0 8 ? * SAT", 8.hours, 90.days)
domain.mediator.set_schedule("0 0 8 ? * SAT", 8.hours, 90.days)
```

Refer to the cron specification to customize the pruning schedule. Here are a few examples:

```
set_schedule("0 0 20 * * ?", 2.hours, retention) // run every evening at 8pm GMT
 ↪for two hours
set_schedule("0 /5 * * * ?", 1.minute, retention) // run every 5 minutes for one
 ↪minute
set_schedule("0 0 0 31 12 ? 2023", 1.day, retention) // run for one specific day
```

For the maximum duration to specify a reliable pruning window end time , the leading fields of the cron expression should should not be wildcards (*) as illustrated in the examples above. If the hour field is fixed, so should the fields for minute and second.

1.33.3.2 Monitoring Pruning Progress

Monitor the pruning state to determine that the pruning schedule allows participant, mediator, and sequencer pruning to keep up with ledger growth, and is not stuck for one of the reasons described below in the Common Notes .

Specifically, monitor the *max-event-age* metrics describing the age of the oldest, un-pruned event (in hours):

```
<participant>.prune.max-event-age
<mediator>.max-event-age
<sequencer>.max-event-age
```

The *max-event-age* metrics should not exceed the value of the pruning schedule *retention* plus the length of the interval. For example, if your schedule specifies a retention of 30 days and a cron that

calls for weekly pruning, *max-event-age* must remain below 37 days. If for any node the *max-event-age* metric exceeds this upper limit, you should consider allocating more time for pruning by reducing the interval between pruning windows or by increasing the *maximum duration* pruning schedule setting.

1.33.3.3 Best Practices

Pruning deletes data from the database, freeing up space, but it does not perform any database maintenance such as table resizing. PostgreSQL supports automatic and manual vacuuming for this purpose, but Oracle lacks such support and relies on the database administrator to manage freed database space.

Pruning is an IO-heavy operation and would have an impact on overall system performance (lowering throughput during pruning by as much as 50% in our test environments), hence it is preferable to schedule pruning during maintenance windows such as after taking database backups or during low load.

A catastrophic failure of a participant and its backup can be mitigated by rebuilding its state from the sequencer by replaying messages. However, this becomes impossible once the required messages have been pruned from the sequencer. For this reason, the backup strategy for participant nodes should be coordinated with the sequencer's pruning schedule.

For high availability nodes that share a common database, the pruning schedule has to be set on an active replica (participant, mediator) or one active shard (database sequencer).

Participants, mediators, and sequencers also expose manual *prune** methods that come with pitfalls. The methods might appear to be hanging unless the range of events and messages specified for pruning is not broken up into sufficiently small chunks. In addition, these manual methods have no built-in mechanism to resume on another node after a high-availability failover. Automatic pruning is recommended instead.

1.33.3.4 Current Limitations

The sequencer will only allow pruning of acknowledged events. As such, if a client such as a participant or mediator stops acknowledging events that have been sent to it, sequencer pruning will be blocked. This is a current limitation.

Generally, idle sequencer clients will only acknowledge once they have observed subsequent events. This means that idle clients normally won't acknowledge the last event. Since each member requests a time-proof once per day, there is a default limitation that a system with idle but connected clients cannot be pruned with a retention window of less than 24 hours. As the topology manager connects to the sequencer but is often idle and only invoked on topology changes, this limitation manifests itself when pruning test environments where aggressive pruning windows of less than 24 hours are used. This can be fixed by adjusting the domain-tracker time of the topology manager: `canton.domains.mydomain.time-tracker.min-observation-duration = 1h`.

Pruning of participants requires the participant to have received a commitment from each counter-participant with which it shares a contract. If a participant becomes defunct and stops sending commitments, pruning of the participant will not work. Therefore, before you disable a participant, please make sure that is not involved in any contract. This is a current limitation.

Refer to the Participant Pruning section to learn how pruning affects Daml applications and the Ledger API.

1.33.4 Participant Pruning

The Daml Ledger API exposes an append-only ledger model; on the other hand, Daml Participants must be able to operate continuously for an indefinite amount of time on a limited amount of hot storage.

In addition, privacy demands[1] may require removing Personally Identifiable Information (PII) upon request.

To satisfy these requirements, the *Pruning Service* Ledger API endpoint[2] allows Daml Participants to support pruning of Daml contracts and transactions that were respectively archived and submitted before or at a given ledger offset.

Please refer to the specific Daml driver information for details about its pruning support.

1.33.4.1 Impacts on Daml Applications

When supported, pruning can be invoked by an operator with administrative privileges at any time on a healthy Daml participant; furthermore, it doesn't require stopping nor suspending normal operation.

Still, Daml applications may be affected in the following ways:

Pruning is potentially a long-running operation and demanding one in terms of system resources; as such, it may significantly reduce Daml Ledger API throughput and increase latency while it is being performed. It is thus strongly recommended to plan pruning invocations, preferably, when the system is offline or at least when very low system utilization is expected.
Pruning may degrade the behavior of or abort in-progress requests if the pruning offset is too recent. In particular, the system might misbehave if command completions are pruned before the command trackers are able to process the completions.
Command deduplication and command tracker retention should always be configured so that the associated windows don't overlap with the pruning window to ensure that their operation is unaffected by pruning.
Pruning may affect the behavior of Ledger API calls that allow to read data from the ledger: see the next sub-section for more information about API impacts.
Pruning of all divulged contracts (see *Prune Request*) does not preserve application visibility over contracts divulged up to the pruning offset, hence applications making use of pruned divulged contracts might start experiencing failed command submissions: see the section below for determining a suitable pruning offset.

Warning: Participants may know of contracts for which they don't know the current activeness status. This happens through *divulgence* where a party learns of the existence of a contract without being guaranteed to ever see its archival. Such contracts are pruned by the feature described on this page as not doing so could easily lead to an ever growing participant state.

During command submission, parties can fetch divulged contracts. This is incompatible with the pruning behaviour described above which allows participant operators to reclaim storage space by pruning divulged contracts. Daml code running on pruned participants should therefore never rely

[1] For example, as enabled by provisions about the right to be forgotten of legislation such as EU's GDPR.
[2] Invoking the Pruning Service requires administrative privileges.

on existence of divulged contracts prior to or at the pruning offset. Instead, such applications MUST ensure re-divulgence of the used contracts.

1.33.4.2 How the Daml Ledger API is Affected

Pruning of old data is not noticed by applications that are up to date. However pruning data in active use by applications can result in the following errors:

> Active data streams from the Daml Participant may abort and need to be re-established by the Daml application from a later offset than pruned, even if they are already streaming past it.
> Requesting information at offsets that predate pruning, including from the ledger's start, will result in a `FAILED_PRECONDITION` gRPC error. - As a consequence, after pruning, a Daml application must bootstrap from the Active Contract Service and a recent offset[3].

Submission validation and Daml Ledger API endpoints that write to the ledger are generally not affected by pruning; an exception is that in-progress calls could abort while awaiting completion.

Please refer to the *protobuf documentation of the API* for details about the `prune` operation itself and the behavior of other Daml Ledger API endpoints when pruning is being or has been performed.

1.33.4.3 Other Limitations

> Pruning may be rejected even if the node is running correctly (for example, to preserve non-repudiation properties); in this case, the application might not be able to archive contracts containing PII or pruning of these contracts may not be possible; thus, actually deleting this PII may also be technically unfeasible.
> Pruning may leave parties, packages, and configuration data on the participant node, even if they are no longer needed for transaction processing, and even if they contain PIIPage 1326, 3.
> Pruning does not move pruned information to cold storage but simply deletes pruned data; for this reason, it is advisable to back up the Participant Index DB before invoking pruning. See the next sub-section for more Participant Index DB-related advice before and after invoking *prune*.
> Pruning is not selective but rather effectively truncates the ledger, removing events on behalf of archived contracts and command completions at the pruning offset and all previous offsets.

1.33.4.4 How Pruning Affects Index DB Administration

Pruning deletes data from the participant's database and therefore frees up space within it, which can and will be reused during the continued operation of the Index DB. Whether this freed up space is handed back to the OS depends on the database in use. For example, in PostgreSQL the deleted data frees up space in the table storage itself, but does not shrink the size of the files backing the tables of the IndexDB. Please refer to the PostgreSQL documentation on *VACUUM* and *VACUUM FULL* for more information.

Activities to be carried out *before* invoking a pruning operation should thus include backing up the Participant Index DB, as pruning will not move information to cold storage but rather it will delete events on behalf of archived contracts and command completions before or at the pruning offset.

In addition, activities to be carried out *after* invoking a pruning operation might include:

[3] This might be improved in future versions.

On a PostgreSQL Index DB, especially if auto-vacuum tuning has not been performed, issuing *VACUUM* commands at appropriate times may improve performance and storage usage by letting the database reuse freed space. Note that *VACUUM FULL* commands are still needed for the OS to reclaim disk space previously used by the database.

Backing up and vacuuming, in addition to pruning itself, are also long-running and resource-hungry operations that might negatively affect the performance of regular workloads and even the availability of the system: this is true in particular for *VACUUM FULL* in PostgreSQL and equivalent commands in other DBMSs. These operations should thus be planned and taken carefully into account when sizing system resources. They should also be scheduled sensibly in relation to the desired sustained performance levels of regular workloads and to the hot storage usage goals.

Professional advice on database administration is strongly recommended that would take into account the DB specifics as well as all of the above aspects.

1.33.4.5 Determine a Suitable Pruning Offset

The *Transaction Service* and the *Active Contract Service* provide offsets of the ledger end of the Transactions, and of Active Contracts snapshots respectively. Such offsets can be passed unchanged to *prune* calls, as long as they are lexicographically lower than the current ledger end. An additional constraint imposed by Canton is that the participant you are pruning must have already exchanged the ACS commitments with other participants for the offset that you prune at. Refer to Canton pruning documentation for more information.

When pruning all divulged contracts, the participant operator can choose the pruning offset (provided that the suitable ACS commitments have already been exchanged) as follows:

Just before the ledger end, if no application hosted on the participant makes use of divulgence
OR
An offset old enough (e.g. older than an arbitrary multi-day grace period) that it ensures that pruning does not affect any recently-divulged contract needed by the applications hosted on the participant.

Scheduled jobs, applications and/or operator tools can be built on top of the Daml Ledger API to implement pruning automatically, for example at regular intervals, or on-demand, for example according to a user-initiated process.

For instance, pruning at regular intervals could be performed by a cron job that:

1. If a pruning interval has been saved to a well-known location:
   a. Backs up the Daml Participant Index DB.
   b. Performs pruning.
   c. (If using PostgreSQL) Performs a *VACUUM FULL* command on the Daml Participant Index DB.
2. Queries the current ledger end and saves its offset.

The interval between 2 (i.e. saving a recent ledger end offset) and the next cron job run determines the data retention window, that should be long enough not to affect deduplication and commands completion. For example, pruning at a recent ledger end offset could be problematic and should be avoided.

Pruning could also be initiated on-demand at the offset of a specific transaction[4], for example as provided by a user application based on search.

[4] Note that all the events on behalf of archived contracts and command completions found at earlier offsets will also be pruned.

1.33.5 Participant Metering

Participant metering is a way to report how many events have been submitted in a given period of time.

Daml command execution results in a Daml transaction that contains events associated with the processing of the command.

The events included in the report include:

Contract creation
Exercise of a contract (including non-consuming exercises and exercise by key)
Fetch of a contract (including fetch by key)
Lookup by contract key

Only events that originated from the local participant are included in the metering. Events received by the local participant from remote participants are *not* included.

Only events contained in committed transactions are included, a failed transaction has no effect on ledger metering.

1.33.5.1 Generate a Metering Report

A metering report is generated using the *Daml assistant* utility.

To run a metering report `daml ledger metering-report` is used with the following metering specific arguments:

`--from` A start date that is used to initiate the reporting period. Events on or after this date will be included.
`--to` An end date that may be used to terminate the reporting period. Events prior to this date will be included. If an end date is not provided then the report will contain counts of all events that occurred on or after the `--from` date.
`--application` Optionally, provide an application to limit the report to that application.

The from and to dates above should be formatted `yyyy-mm-dd`. The exact timestamp used for the report will be the start of the UTC day provided.

Ledger metering is not affected by participant pruning.

Other non-metering specific Daml assistant flags may also be used alongside those shown above.

1.33.5.2 Example

To report on all applications for January 2022 the following from/to flags would be set:

```
daml ledger metering-report --from 2022-01-01 --to 2022-02-01
```

1.33.5.3 Output

```
{
    "participant": "some-participant",
    "request": {
        "from": "2022-01-01T00:00:00Z",
        "to": "2022-02-01T00:00:00Z"
    },
    "final": true,
    "applications": [
        {
            "application": "some-application",
            "events": 42
        }
    ],
    "check": {
        "digest": "sxRZw40JJ5gWGUJoecm6-i-UPQ2imBVqeOYnbmYhVNA=",
        "scheme": "canton-enterprise-2022"
    }
}
```

The output consists of the following sections:

participant The name of the local participant the report applies to

request This section gives details of the parameters that were used to generate the report

final This field will be set to `true` if a `--to` date was provided and the `--to` date is in the past. Once a report is marked as final the event counts will never change and so may be used for billing purposes.

applications This section will give an event count for each application used in the reporting period.

check This section is used by the billing operator to verify that the report has not been modified.

1.33.6 API Configuration

A domain node exposes two main APIs: the admin-api and the public-api, while the participant node exposes the ledger-api and the admin-api. In this section, we will explain what the APIs do and how they can be configured.

1.33.6.1 Default Ports

Canton assigns ports automatically for all the APIs of all the configured nodes if the port has not been configured explicitly. The ports are allocated according to the following scheme:

```
/** Participant node default ports */
val ledgerApiPort = defaultPortStart(4001)
val participantAdminApiPort = defaultPortStart(4002)

/** Domain node default ports */
val domainPublicApiPort = defaultPortStart(4201)
val domainAdminApiPort = defaultPortStart(4202)

/** External sequencer node default ports (enterprise-only) */
```

(continues on next page)

(continued from previous page)

```
val sequencerPublicApiPort = defaultPortStart(4401)
val sequencerAdminApiPort = defaultPortStart(4402)

/** External mediator node default port (enterprise-only) */
val mediatorAdminApiPort = defaultPortStart(4602)

/** Domain node default ports */
val domainManagerAdminApiPort = defaultPortStart(4801)

/** External sequencer node x default ports (enterprise-only) */
val sequencerXPublicApiPort = defaultPortStart(5001)
val sequencerXAdminApiPort = defaultPortStart(5002)

/** External mediator node x default port (enterprise-only) */
val mediatorXAdminApiPort = defaultPortStart(5202)

/** Increase the default port number for each new instance by portStep */
private val portStep = 10
```

1.33.6.2 Administration API

The nature and scope of the admin api on participant and domain nodes has some overlap. As an example, you will find the same key management commands on the domain and the participant node API, whereas the participant has different commands to connect to several domains.

The configuration currently is simple (see the TLS example below) and just takes an address and a port. The address defaults to `127.0.0.1` and a default port is assigned if not explicitly configured.

You should not expose the admin-api publicly in an unsecured way as it serves administrative purposes only.

1.33.6.3 TLS Configuration

Both, the Ledger API and the admin API provide the same TLS capabilities and can be configured using the same configuration directives. TLS provides end-to-end channel encryption between the server and client, and depending on the settings, server or mutual authentication. Ensure that the required TLS system dependencies are installed, e.g., *libcrypt1* on Ubuntu.

A full configuration example is given by

```
canton.participants.participant4.ledger-api {
  address = "127.0.0.1" // IP / DNS must be SAN of certificate to allow local
↪connections from the canton process
  port = 5041
  tls {
    // the certificate to be used by the server
    cert-chain-file = "./tls/participant.crt"
    // private key of the server
    private-key-file = "./tls/participant.pem"
    // trust collection, which means that all client certificates will be
↪verified using the trusted
    // certificates in this store. if omitted, the JVM default trust store is
↪used.
```

(continues on next page)

(continued from previous page)

```
    trust-collection-file = "./tls/root-ca.crt"
    // define whether clients need to authenticate as well (default not)
    client-auth = {
      // none, optional and require are supported
      type = require
      // If clients are required to authenticate as well, we need to provide a␣
↪client
      // certificate and the key, as Canton has internal processes that need to␣
↪connect to these
      // APIs. If the server certificate is trusted by the trust-collection, then␣
↪you can
      // just use the server certificates. Otherwise, you need to create separate␣
↪ones.
      admin-client {
        cert-chain-file = "./tls/admin-client.crt"
        private-key-file = "./tls/admin-client.pem"
      }
    }
    // minimum-server-protocol-version = ...
    // ciphers = ...
  }
}
```

These TLS settings allow a connecting client to ensure that it is talking to the right server. In this example, we have also enabled client authentication, which means that the client needs to present a valid certificate (and have the corresponding private key). The certificate is valid if it has been signed by a key in the trust store.

The `trust-collection-file` allows us to provide a file based trust store. If omitted, the system will default to the built-in `JVM` trust store. The file must contain all client certificates (or parent certificates which were used to sign the client certificate) who are trusted to use the API. The format is just a collection of PEM certificates (in the right order or hierarchy), not a java based trust store.

In order to operate the server just with server-side authentication, you can just omit the section on `client-auth`. However, if `client-auth` is set to `require`, then Canton also requires a client certificate, as various Canton internal processes will connect to the process itself through the API.

All the private keys need to be in the `pkcs8` PEM format.

By default, Canton only uses new versions of TLS and strong ciphers. You can also override the default settings using the variables `ciphers` and `protocols`. If you set these settings to `null`, the default JVM values will be used.

Note: Error messages on TLS issues provided by the networking library `netty` are less than optimal. If you are struggling with setting up TLS, please enable `DEBUG` logging on the `io.netty` logger.

Note that the configuration hierarchy for a remote participant console is slightly different from the in-process console or participant shown above. For configuring a remote console with TLS, please see the scaladocs for a TlsClientConfig (see also *how scaladocs relates to the configuration*).

If you need to create a set of testing TLS certificates, you can use the following openssl commands:

```
DAYS=3650

function create_key {
  local name=$1
  openssl genrsa -out "${name}.key" 4096
  # netty requires the keys in pkcs8 format, therefore convert them appropriately
  openssl pkcs8 -topk8 -nocrypt -in "${name}.key" -out "${name}.pem"
}

# create self signed certificate
function create_certificate {
  local name=$1
  local subj=$2
  openssl req -new -x509 -sha256 -key "${name}.key" \
              -out "${name}.crt" -days ${DAYS} -subj "$subj"
}

# create certificate signing request with subject and SAN
# we need the SANs as our certificates also need to include localhost or the
# loopback IP for the console access to the admin-api and the ledger-api
function create_csr {
  local name=$1
  local subj=$2
  local san=$3
  (
    echo "authorityKeyIdentifier=keyid,issuer"
    echo "basicConstraints=CA:FALSE"
    echo "keyUsage = digitalSignature, nonRepudiation, keyEncipherment,
↪dataEncipherment"
  ) > ${name}.ext
  if [[ -n $san ]]; then
    echo "subjectAltName=${san}" >> ${name}.ext
  fi
  # create certificate (but ensure that localhost is there as SAN as otherwise,
↪admin local connections won't work)
  openssl req -new -sha256 -key "${name}.key" -out "${name}.csr" -subj "$subj"
}

function sign_csr {
  local name=$1
  local sign=$2
  openssl x509 -req -sha256 -in "${name}.csr" -extfile "${name}.ext" -CA "${sign}.
↪crt" -CAkey "${sign}.key" -CAcreateserial  \
               -out "${name}.crt" -days ${DAYS}
  rm "${name}.ext" "${name}.csr"
}

function print_certificate {
  local name=$1
  openssl x509 -in "${name}.crt" -text -noout
}

# create root certificate
create_key "root-ca"
create_certificate "root-ca" "/O=TESTING/OU=ROOT CA/
↪emailAddress=canton@digitalasset.com"
```

(continues on next page)

(continued from previous page)

```
#print_certificate "root-ca"

# create domain certificate
create_key "domain"
create_csr "domain" "/O=TESTING/OU=DOMAIN/CN=localhost/
↪emailAddress=canton@digitalasset.com" "DNS:localhost,IP:127.0.0.1"
sign_csr "domain" "root-ca"
print_certificate "domain"

# create participant certificate
create_key "participant"
create_csr "participant" "/O=TESTING/OU=PARTICIPANT/CN=localhost/
↪emailAddress=canton@digitalasset.com" "DNS:localhost,IP:127.0.0.1"
sign_csr "participant" "root-ca"

# create participant client key and certificate
create_key "admin-client"
create_csr "admin-client" "/O=TESTING/OU=ADMIN CLIENT/CN=localhost/
↪emailAddress=canton@digitalasset.com"
sign_csr "admin-client" "root-ca"
print_certificate "admin-client"
```

If you are having problems with SSL connectivity, you can enable SSL debugging by adding the following flag to the command line when starting Canton:

```
bin/canton -Djavax.net.debug=all
```

This will enable detailed SSL debugging information to be printed to the console, which can help you diagnose and troubleshoot any problems you may be experiencing. It is recommended to only enable this flag when needed, as the output can be very verbose and may impact the performance of your application.

1.33.6.4 Keep Alive

In order to prevent load-balancers or firewalls from terminating long running RPC calls in the event of some silence on the connection, all gRPC connections enable keep-alive by default. An example configuration for an adjusted setting is given below:

```
canton.participants.participant2 {
  admin-api {
    address = "127.0.0.1"
    port = 5022
    keep-alive-server {
      time = 40s
      timeout = 20s
      permit-keep-alive-time = 20s
    }
  }
  sequencer-client {
    keep-alive-client {
      time = 60s
      timeout = 30s
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

gRPC client connections are configured with `keep-alive-client`, with two settings: `time`, and `timeout`. The effect of the `time` and `timeout` settings are described in the gRPC documentation.

Servers can additionally change another setting: `permit-keep-alive-time`. This specifies the most aggressive keep-alive time that a client is permitted to use. If a client uses keep-alive `time` that is more aggressive than the `permit-keep-alive-time`, the connection will be terminated with a GOAWAY frame with `too_many_pings` as the debug data. This setting is described in more detail in the gRPC documentation and gRPC manual page.

1.33.6.5 Max Inbound Message Size

The APIs exposed by both the participant (ledger API and admin API) as well as by the domain (public API and admin API) have an upper limit on incoming message size. To increase this limit to accommodate larger payloads, the flag `max-inbound-message-size` has to be set for the respective API to the maximum message size in **bytes**.

For example, to configure a participant's ledger API limit to 20MB:

```
canton.participants.participant2.ledger-api {
  address = "127.0.0.1"
  port = 5021
  max-inbound-message-size = 20971520
}
```

1.33.6.6 Participant Configuration

Ledger API

The configuration of the ledger API is similar to the admin API configuration, except that the group starts with `ledger-api` instead of `admin-api`.

JWT Authorization

The Ledger API supports JWT based authorization checks as described in the *Authorization documentation*.

In order to enable JWT authorization checks, your safe configuration options are

```
_shared {
  ledger-api {
    auth-services = [{
      // type can be
      //   jwt-rs-256-crt
      //   jwt-es-256-crt
      //   jwt-es-512-crt
      type = jwt-rs-256-crt
      // we need a certificate file (abcd.cert)
```

(continues on next page)

(continued from previous page)

```
      certificate = ${JWT_CERTIFICATE_FILE}
    }]
  }
}
```

`jwt-rs-256-crt`. The participant will expect all tokens to be signed with RS256 (RSA Signature with SHA-256) with the public key loaded from the given X.509 certificate file. Both PEM-encoded certificates (text files starting with `-----BEGIN CERTIFICATE-----`) and DER-encoded certificates (binary files) are supported.
`jwt-es-256-crt`. The participant will expect all tokens to be signed with ES256 (ECDSA using P-256 and SHA-256) with the public key loaded from the given X.509 certificate file. Both PEM-encoded certificates (text files starting with `-----BEGIN CERTIFICATE-----`) and DER-encoded certificates (binary files) are supported.
`jwt-es-512-crt`. The participant will expect all tokens to be signed with ES512 (ECDSA using P-521 and SHA-512) with the public key loaded from the given X.509 certificate file. Both PEM-encoded certificates (text files starting with `-----BEGIN CERTIFICATE-----`) and DER-encoded certificates (binary files) are supported.

Instead of specifying the path to a certificate, you can also use a JWKS URL. In that case, the participant will expect all tokens to be signed with RS256 (RSA Signature with SHA-256) with the public key loaded from the given JWKS URL.

```
_shared {
  ledger-api {
    auth-services = [{
      type = jwt-rs-256-jwks
      // we need a URL to a jwks key, e.g. https://path.to/jwks.key
      url = ${JWT_URL}
    }]
  }
}
```

Warning: For testing purposes only, you can also specify a shared secret. In that case, the participant will expect all tokens to be signed with HMAC256 with the given plaintext secret. This is not considered safe for production.

```
_shared {
  ledger-api {
    auth-services = [{
      type = unsafe-jwt-hmac-256
      secret = "not-safe-for-production"
    }]
  }
}
```

Note: To prevent man-in-the-middle attacks, it is highly recommended to use TLS with server authentication as described in *TLS Configuration* for any request sent to the Ledger API in production.

Note that you can define multiple authorization plugins. If more than one is defined, the system will use the claim of the first auth plugin that does not return Unauthorized.

If no authorization plugins are defined, a default (wildcard) authorization method is used. Under it, all valid ledger API requests are accepted without the system performing any request authorization. To explicitly define the default authorization method, use the following configuration:

```
_shared {
  ledger-api {
    auth-services = [{
      type = wildcard
    }]
  }
}
```

Leeway Parameters for JWT Authorization

You can define leeway parameters for authorization using JWT tokens. An authorization which fails due to clock skew between the signing and the verification of the tokens can be eased by specifying a leeway window in which the token should still be considered valid. Leeway can be defined either specifically for the **Expiration Time (exp)**, **Not Before (nbf)** and **Issued At (iat)** claims of the token or by a default value for all three. The values defining the leeway for each of the three specific fields override the default value if present. The leeway parameters should be given in seconds and can be defined as in the example configuration below:

```
_shared {
  parameters.ledger-api-server-parameters.jwt-timestamp-leeway {
    default = 5
    expires-at = 10
    issued-at = 15
    not-before = 20
  }
}
```

Configuring the Target Audience for JWT Authorization

The default audience (`aud` field in the audience based token) for authenticating on the Ledger API using JWT is `https://daml.com/participant/jwt/aud/participant/${participantId}`. Other audiences can be configured explicitly using the custom target audience configuration option:

```
canton {
  participants {
    participant {
      ledger-api {
        auth-services = [{
          type = jwt-rs-256-jwks
          url = "https://target.audience.url/jwks.json"
          target-audience = "https://rewrite.target.audience.url"
        }]
      }
    }
  }
}
```

1.33.6.7 Domain Configurations

Public API

The domain configuration requires the same configuration of the `admin-api` as the participant. Next to the `admin-api`, we need to configure the `public-api`, which is the api where all participants connect.

Authentication Token

Authentication of the restricted services is built into the public sequencer api, leveraging the participant signing keys. You don't need to do anything in order to set this up; it is enforced automatically and can't be turned off. The same mechanism is used to check the authentication of the domain topology manager and the mediator.

The token is generated during the handshake between the node and the sequencer. By default, it is valid for one hour. The nodes automatically renew the token in the background before it expires. The lifetime of the tokens and of the nonce can be reconfigured using

```
canton.domains.mydomain.public-api {
    token-expiration-time = 60m
    nonce-expiration-time = 1m
}
```

However, we suggest keeping the default values.

TLS Encryption

As with the admin-api, network traffic can (and should) be encrypted using TLS. This is particularly crucial for the Public API.

An example configuration section which enables TLS encryption and server-side TLS authentication is given by

```
canton.domains.acme.public-api {
  port = 5028
  address = localhost // defaults to 127.0.0.1
  tls {
    cert-chain-file = "./tls/domain.crt"
    private-key-file = "./tls/domain.pem"
    // minimum-server-protocol-version = TLSv1.3, optional argument
    // ciphers = null // use null to default to JVM ciphers
  }
}
```

If TLS is used on the server side with a self-signed certificate, we need to pass the certificate chain during the connect call of the participant. Otherwise, the default root certificates of the Java runtime will be used. An example would be:

```
participant3.domains.connect(
  domainAlias = "acme",
  connection = s"https://$hostname:$port",
```

(continues on next page)

(continued from previous page)

```
  certificatesPath = certs, // path to certificate chain file (.pem) of server
)
```

1.33.6.8 Usage of native libraries by Netty

Canton ships with native libraries (for some processor architectures: x86_64, ARM64, S390_64) so that the Netty network access library can take advantage of `epoll` on Linux, generally leading to improved performance and less pressure on the JVM garbage collector. The available native is picked up automatically and it falls back to the standard NIO library if running on unsupported operating systems or architectures.

Furthermore, the usage of the native library can also be switched off by setting the following: *-Dio.netty.transport.noNative=true*. Even when this is expected, falling back to NIO might lead to a warning being emitted at *DEBUG* level on your log.

1.33.7 Sequencer Connections

Any member of a Canton network, whether a participant, mediator or topology manager, connects to the domain by virtue of connecting to a sequencer of that domain (there can be multiple thereof). The component managing this connection is called the `SequencerClient`.

A participant can connect to multiple domains (preview) simultaneously, but a mediator or topology manager will only connect to a single domain. Therefore, managing the sequencer connections of a participant differs slightly from managing a mediator or topology manager connection.

In the following sections, we will explain how to manage such sequencer connections.

1.33.7.1 Participant Connections

The *domain connectivity commands* allow the administrator of a Canton node to manage connectivity to domains. Generally, the key command to add new connections is given by the *register command*. While this is the command with the broadest ability to configure the connection, there are a few convenience macros that combine a series of steps to simplify administrative operations.

Connect Using Macros

Connect to Local Sequencers

When a participant should connect to a sequencer or domain that is running in the same process, you can use the `domains.connect_local` macro and simply provide the reference to the local node.

```
@ participant1.domains.connect_local(sequencer1)
```

The `connect_local` macro will generate the appropriate configuration settings for the provided sequencer and instruct the participant to connect to it using the `register` command.

Please note that you can also pass a local `DomainReference` to the `connect_local` call in case you are running an embedded domain.

Connect to Remote Sequencers

If you are connecting to a sequencer that is running on a remote host, you need to know the address and port the sequencer is configured to listen to. You can print out the port the sequencer is listening to using:

```
@ sequencer1.config.publicApi.port
res2: Port = Port(n = 30123)
```

You can also check that the address is set such that remote processes can connect to it:

```
@ sequencer1.config.publicApi.address
res3: String = "0.0.0.0"
```

By default, a sequencer will listen to `127.0.0.1`, which is localhost. These is a safe default and it means that only processes running locally can connect to the sequencer (it is also set by default for the Ledger API and the Admin API). If you want to support remote processes connecting to the given sequencer, you need to explicitly configure it using:

```
// enable access of remote processes to the sequencer
canton.sequencers.sequencer1.public-api.address = 0.0.0.0
```

In this example, `sequencer1` and `sequencer2` are configured without TLS, whereas `sequencer3` is configured to use TLS:

```
@ sequencer3.config.publicApi.tls
res4: Option[TlsBaseServerConfig] = Some(
  value = TlsBaseServerConfig(
    certChainFile = ExistingFile(file = ./tls/sequencer3-127.0.0.1.crt),
    privateKeyFile = ExistingFile(file = ./tls/sequencer3-127.0.0.1.pem),
    minimumServerProtocolVersion = Some(value = "TLSv1.2"),
    ciphers = Some(
      value = List(
        "TLS_AES_256_GCM_SHA384",
        "TLS_CHACHA20_POLY1305_SHA256",
        "TLS_AES_128_GCM_SHA256",
        "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384",
        "TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256"
      )
    )
  )
)
```

To connect to `sequencer3` using the `connect` macro, we need to create an URL string:

```
@ val port = sequencer3.config.publicApi.port
port : Port = Port(n = 30119)
```

```
@ val url = s"https://127.0.0.1:${port}"
url : String = "https://127.0.0.1:30119"
```

Please note that you need to adjust the `https` to `http` if you are not using `TLS` on the public sequencer Api. If the sequencer is using TLS certificates (e.g. self-signed) that cannot be automatically validated using your JVMs trust store, you have to provide the custom certificate such that the client can verify the sequencer public API TLS certificate. Let's assume that this root certificate is given by:

```
@ val certificatesPath = "tls/root-ca.crt"
certificatesPath : String = "tls/root-ca.crt"
```

You can now connect the participant to the sequencer using:

```
@ participant2.domains.connect("mydomain", connection = url, certificatesPath =
↪certificatesPath)
res8: DomainConnectionConfig = DomainConnectionConfig(
  domain = Domain 'mydomain',
  sequencerConnections = Sequencer 'DefaultSequencer' -> GrpcSequencerConnection(
    endpoints = https://127.0.0.1:30119,
    transportSecurity = true,
    customTrustCertificates = Some(2d2d2d2d2d42)
  ),
  manualConnect = false,
  domainId = None(),
  priority = 0,
  initialRetryDelay = None(),
  maxRetryDelay = None()
)
```

Connect to High-Availability Sequencers

Important: This feature is only available in Canton Enterprise

The Daml Enterprise version of Canton lets you connect a participant to multiple sequencers for the purpose of high availability. If one sequencer shuts down, the participant will then automatically fail over to the second sequencer.

Such a connection can be configured using the `connect_multi`:

```
@ participant3.domains.connect_multi("mydomain", Seq(sequencer1, sequencer2))
res9: DomainConnectionConfig = DomainConnectionConfig(
  domain = Domain 'mydomain',
  sequencerConnections = Sequencer 'DefaultSequencer' -> GrpcSequencerConnection(
    endpoints = Seq(http://0.0.0.0:30123, http://127.0.0.1:30121),
    transportSecurity = false,
    customTrustCertificates = None()
  ),
  manualConnect = false,
  domainId = None(),
  priority = 0,
  initialRetryDelay = None(),
  maxRetryDelay = None()
)
```

In such a setting, if a sequencer node goes down, the participant will round-robin through the available list of sequencers. The *reference documentation* provides further information on how to connect to highly available sequencers, and the *high availability guide* has instructions on how to set up highly available domains.

Currently, all the sequencer connections used by a node need to be using TLS or not. A mixed mode where one sequencer is using TLS and another not is not supported.

Connect Using Register

The highest level of control over your domain connection is given by using `register` with a configuration of type `DomainConnectionConfig`. By default, the connection configuration only requires two arguments: the domain alias and the connection URL. In this guide, we'll cover all arguments.

First, we need to associate the domain connection to an alias. An alias is an arbitrary name chosen by the operator of the participant to manage the given connection:

```
@ val domainAlias = "mydomain"
domainAlias : String = "mydomain"
```

A domain alias is just a string wrapped into the type `DomainAlias`. This is done implicitly in the console, which allows you to use a string instead.

Next, you need to create a connection description of type `SequencerConnection`. The public sequencer API in Canton is based on gRPC, which uses `HTTP 2.0`. In this example, we build the URLs by inspecting the configurations:

```
@ val urls = Seq(sequencer1, sequencer2).map(_.config.publicApi.port).map(port =>
 ↪s"http://127.0.0.1:${port}")
urls : Seq[String] = List("http://127.0.0.1:30123", "http://127.0.0.1:30121")
```

However, the url can also be entered as a string. A connection is then built using:

```
@ val sequencerConnectionWithoutHighAvailability = com.digitalasset.canton.
 ↪sequencing.GrpcSequencerConnection.tryCreate(urls(0))
sequencerConnectionWithoutHighAvailability : GrpcSequencerConnection =
 ↪GrpcSequencerConnection(
  endpoints = http://127.0.0.1:30123,
  transportSecurity = false,
  customTrustCertificates = None()
)
```

A second sequencer URL can be added using:

```
@ val sequencerConnection = sequencerConnectionWithoutHighAvailability.
 ↪addEndpoints(urls(1))
sequencerConnection : SequencerConnection = GrpcSequencerConnection(
  endpoints = Seq(http://127.0.0.1:30123, http://127.0.0.1:30121),
  transportSecurity = false,
  customTrustCertificates = None()
)
```

While the `connect` macros allow you to pass in a file path as an argument for the optional TLS certificate, you need to resolve this argument and load the certificate into a `ByteString` when working with `GrpcSequencerConnection`. There is a utility function that allows you to read a certificate from a file into a ByteString such that it can be used to create an appropriate sequencer connection:

```
@ val certificate = com.digitalasset.canton.util.BinaryFileUtil.
 ↪tryReadByteStringFromFile("tls/root-ca.crt")
certificate : com.google.protobuf.ByteString = <ByteString@5ca338e4 size=1960
 ↪contents="-----BEGIN CERTIFICATE-----\nMIIFeTCCA2GgAwIBAgI...">
```

```
@ val connectionWithTLS = com.digitalasset.canton.sequencing.
↪GrpcSequencerConnection.tryCreate("https://daml.com", customTrustCertificates =
↪Some(certificate))
connectionWithTLS : GrpcSequencerConnection = GrpcSequencerConnection(
  endpoints = https://daml.com:443,
  transportSecurity = true,
  customTrustCertificates = Some(2d2d2d2d2d42)
)
```

Next, you can assign a priority to the domain by setting the priority parameter:

```
@ val priority = 10 // default is 0 if not set
priority : Int = 10
```

This parameter is used to determine the domain to which a transaction should be sent if there are multiple domains connected (early access feature). The domain with the highest priority that can run a certain transaction will be picked.

Finally, when configuring a domain connection, the parameter `manualConnect` can be used when the domain should not be auto-reconnected on startup. By default, you would set:

```
@ val manualConnect = false
manualConnect : Boolean = false
```

If a domain connection is configured to be manual, it will not reconnect automatically on startup; it has to be reconnected specifically using:

```
@ participant3.domains.reconnect("mydomain")
res18: Boolean = true
```

Very security sensitive users that do not trust TLS to check for authenticity of the sequencer API can additionally pass an optional `domainId` of the target domain into the configuration. In this case, the participant will check that the sequencer it is connecting to can produce the cryptographic evidence that it actually is the expected domain. The domainId can be obtained from the domain manager:

```
@ val domainId = Some(domainManager1.id)
domainId : Some[DomainId] = Some(value = domainManager1::1220b9035a1f...)
```

These parameters together can be used to define a connection configuration:

```
@ val config = DomainConnectionConfig(domain = "mydomain", sequencerConnection,
↪manualConnect, domainId, priority)
config : DomainConnectionConfig = DomainConnectionConfig(
  domain = Domain 'mydomain',
  sequencerConnections = Sequencer 'DefaultSequencer' -> GrpcSequencerConnection(
    endpoints = Seq(http://127.0.0.1:30123, http://127.0.0.1:30121),
    transportSecurity = false,
    customTrustCertificates = None()
  ),
  manualConnect = false,
  domainId = Some(domainManager1::1220b9035a1f...),
  priority = 10,
  initialRetryDelay = None(),
  maxRetryDelay = None()
)
```

All other parameters are expert settings and should not be used. The `config` object can now be used to connect a participant to a sequencer:

```
@ participant4.domains.register(config)
```

Inspect Connections

You can inspect the registered domain connections using:

```
@ participant2.domains.list_registered()
res22: Seq[(DomainConnectionConfig, Boolean)] = Vector(
  (
    DomainConnectionConfig(
      domain = Domain 'mydomain',
      sequencerConnections = Sequencer 'DefaultSequencer' ->
↪GrpcSequencerConnection(
        endpoints = https://127.0.0.1:30119,
        transportSecurity = true,
        customTrustCertificates = Some(2d2d2d2d2d42)
      ),
      manualConnect = false,
      domainId = None(),
      priority = 0,
      initialRetryDelay = None(),
      maxRetryDelay = None()
    ),
    true
  )
)
```

You can also get the aliases of the currently connected domains using:

```
@ participant2.domains.list_connected()
res23: Seq[ListConnectedDomainsResult] = Vector(
  ListConnectedDomainsResult(
    domainAlias = Domain 'mydomain',
    domainId = domainManager1::1220b9035a1f...,
    healthy = true
  )
)
```

And you can inspect the configuration of a specific domain connection using:

```
@ participant2.domains.config("mydomain")
res24: Option[DomainConnectionConfig] = Some(
  value = DomainConnectionConfig(
    domain = Domain 'mydomain',
    sequencerConnections = Sequencer 'DefaultSequencer' ->
↪GrpcSequencerConnection(
      endpoints = https://127.0.0.1:30119,
      transportSecurity = true,
      customTrustCertificates = Some(2d2d2d2d2d42)
    ),
    manualConnect = false,
    domainId = None(),
```

(continues on next page)

(continued from previous page)

```
    priority = 0,
    initialRetryDelay = None(),
    maxRetryDelay = None()
  )
)
```

Modify Connections

Domain connection configurations can be updated using the `modify` function:

```
@ participant2.domains.modify("mydomain", _.copy(priority = 20))
```

The second argument is a mapping function which receives as input argument a `DomainConnectionConfig` and needs to return a `DomainConnectionConfig`. Every case class has a default `copy` method that allows overriding arguments.

The modify command on the participant will only take effect after restarting the domain connection explicitly (or restarting the entire node):

```
@ participant2.domains.disconnect("mydomain")
```

```
@ participant2.domains.reconnect("mydomain")
res27: Boolean = true
```

On the mediator and domain manager node, the change is effected immediately.

Update a Custom TLS Trust Certificate

In some cases (in particular in test environments), you might be using self-signed certificates as a root of trust for the TLS sequencer connection. Whenever this root of trust changes, the clients need to update the custom root certificate accordingly.

This can be done through the following steps. First, you need to load the certificate from a file:

```
@ val certificate = com.digitalasset.canton.util.BinaryFileUtil.
↪tryReadByteStringFromFile("tls/root-ca.crt")
certificate : com.google.protobuf.ByteString = <ByteString@6f432dcb size=1960□
↪contents="-----BEGIN CERTIFICATE-----\nMIIFeTCCA2GgAwIBAgI..."
```

This step loads the root certificate from a file and stores it into a variable that can be used subsequently. Next, you create a new connection object, passing in the certificate:

```
@ val connection = com.digitalasset.canton.sequencing.GrpcSequencerConnection.
↪tryCreate(url, customTrustCertificates = Some(certificate))
connection : GrpcSequencerConnection = GrpcSequencerConnection(
  endpoints = https://127.0.0.1:30119,
  transportSecurity = true,
  customTrustCertificates = Some(2d2d2d2d2d42)
)
```

Finally, you update the sequencer connection settings on the participant node:

```
@ participant2.domains.modify("mydomain", _.
↪copy(sequencerConnections=SequencerConnections.single(connection)))
```

For mediators / domain managers, you can update the certificate accordingly.

Enable and Disable Connections

A participant can disconnect from a domain using:

```
@ participant2.domains.disconnect("mydomain")
```

Reconnecting to the domain can be done either on a per domain basis:

```
@ participant2.domains.reconnect("mydomain")
res32: Boolean = true
```

Or for all registered domains that are not configured to require a manual connection:

```
@ participant2.domains.reconnect_all()
```

1.33.7.2 Mediator and Domain Manager

Both the mediator and the domain manager connect to the domain using sequencer connections. The sequencer connections are configured when the nodes are initialized:

```
@ mediator1.mediator.help("initialize")
initialize(domainId: com.digitalasset.canton.topology.DomainId, mediatorId: com.
↪digitalasset.canton.topology.MediatorId, domainParameters: com.digitalasset.
↪canton.admin.api.client.data.StaticDomainParameters, sequencerConnection: com.
↪digitalasset.canton.sequencing.SequencerConnection, topologySnapshot:□
↪Option[com.digitalasset.canton.topology.store.StoredTopologyTransactions[com.
↪digitalasset.canton.topology.transaction.TopologyChangeOp.Positive]]): com.
↪digitalasset.canton.crypto.PublicKey
Initialize a mediator
initialize(domainId: com.digitalasset.canton.topology.DomainId, mediatorId: com.
↪digitalasset.canton.topology.MediatorId, domainParameters: com.digitalasset.
↪canton.admin.api.client.data.StaticDomainParameters, sequencerConnections: com.
↪digitalasset.canton.sequencing.SequencerConnections, topologySnapshot:□
↪Option[com.digitalasset.canton.topology.store.StoredTopologyTransactions[com.
↪digitalasset.canton.topology.transaction.TopologyChangeOp.Positive]]): com.
↪digitalasset.canton.crypto.PublicKey
Initialize a mediator
```

The sequencer connection of a mediator and domain manager can be inspected using:

```
@ mediator1.sequencer_connection.get()
res35: Option[SequencerConnections] = Some(
  value = Sequencer 'DefaultSequencer' -> GrpcSequencerConnection(
    endpoints = http://0.0.0.0:30123,
    transportSecurity = false,
    customTrustCertificates = None()
  )
)
```

In some cases, the connection settings have to be updated. For this purpose, the two following functions can be used. First, the connection information can just be set using:

```
@ mediator1.sequencer_connection.set(sequencer1.sequencerConnection)
```

It can also be amended using:

```
@ mediator1.sequencer_connection.modify(_.addEndpoints(sequencer2.
↪sequencerConnection))
```

Please note that the connection changes immediately, without requiring a restart.

1.33.8 Repairing Nodes

The Canton platform is generally built to self-heal and automatically recover from issues. As such, if there is a situation where some degradation can be expected, there should be some code that yields graceful degradation and automated recovery from said issues.

Common examples are database outages (retry until success) or network outages (failover and reconnect until success).

Canton should report such issues as warnings to alert an operator about the degradation of its dependencies, but generally, should not require any manual intervention to recover from a degradation.

However, not all situations can be foreseen and corruptions of systems can always happen in unanticipated ways. Therefore, Canton can always be manually repaired somehow. This means that whatever the corruption is, there are a series of operational steps that can be made in order to recover the correct state of a node. If several nodes in the distributed system are affected, it may be necessary to coordinate the recovery among the affected nodes.

Conceptually, this means that Canton recovery is structured along the four layers:

1. Automated self-recovery and self-healing.
2. Recovery from crash or restart by re-creating a consistent state from the persisted store.
3. Standard disaster recovery from a database backup in case of database outage and replay from domain.
4. Corruption disaster recovery using repair and other console commands to re-establish a consistent state within the distributed system.

If you run into corruption issues, you need to first understand what caused the issue. Ideally, you can contact our support team to help you diagnose the issue and provide you with a custom recipe on how to recover from your issue (and prevent recurrence).

The toolbox the support engineers have at hand are:

Exporting / importing secret keys
Manually initializing nodes
Exporting / importing DARs
Exporting / importing topology transactions
Manually adding or removing contracts from the active contract set
Moving contracts from one domain to another
Manually ignoring faulty transactions (and then using add / remove contract to repair the ACS).

All these methods are very powerful but dangerous. You should not attempt to repair your nodes on your own as you risk severe data corruption.

Keep in mind that the corruption of the system state may not have been discovered immediately; thus, the corruption may have leaked out through the APIs to the applications using the corrupted node. Bringing the node back into a correct state with respect to the other nodes in the distributed system can therefore make the application state inconsistent with the nodes state. Accordingly, the application should either re-initialize itself from the repaired state or itself offer tools to fix inconsistencies.

1.33.8.1 Preparation

As contracts (1) belong to parties and (2) are instances of Daml templates defined in Daml Archives (DARs), importing contracts to Canton also requires creating corresponding parties and uploading DARs.

> Contracts are often interdependent requiring care to honor dependencies such that the set of imported contracts is internally consistent. This requires particular attention if you choose to modify contracts prior to their import.
> Additionally use of divulgence in the original ledger has likely introduced non-obvious dependencies that may impede exercising contract choices after import. As a result such divulged contracts need to be re-divulged as part of the import (by exercising existing choices or if there are no-side-effect-free choices that re-divulge the necessary contracts by extending your Daml models with new choices).
> Party Ids have a stricter format on Canton than on non-Canton ledgers ending with a required fingerprint suffix, so at a minimum, you will need to remap party ids.
> *Canton contract keys* do not have to be unique, so if your Daml models rely on uniqueness, consider extending the models using *these strategies* or limit your Canton Participants to connect to a single *Canton domain with unique contract key semantics*.
> Canton does not support implicit party creation, so be sure to create all needed parties explicitly.
> In addition you could choose to spread contracts, parties, and DARs across multiple Canton Participants.

With the above requirements in mind, you are ready to plan and execute the following three step process:

1. Download parties and contracts from the existing Daml Participant Node and locate the DAR files that the contracts are based on.
2. Modify the parties and contracts (at the minimum assigning Canton-conformant party ids).
3. Provision Canton Participants along with at least one Canton Domain. Then upload DARs, create parties, and finally the contracts to the Canton participants. Finally connect the participants to the domain(s).

1.33.8.2 Importing an actual Ledger

To follow along with this guide, ensure you have *installed and unpacked the Canton release bundle* and run the following commands from the canton-X.Y.Z directory to set up the initial topology.

```
export CANTON=`pwd`
export CONF="$CANTON/examples/03-advanced-configuration"
export IMPORT="$CANTON/examples/07-repair"
bin/canton \
  -c $IMPORT/participant1.conf,$IMPORT/participant2.conf,$IMPORT/participant3.
↪conf,$IMPORT/participant4.conf \
```

(continues on next page)

(continued from previous page)

```
  -c $IMPORT/domain-export-ledger.conf,$IMPORT/domain-import-ledger.conf \
  -c $CONF/storage/h2.conf,$IMPORT/enable-preview-commands.conf \
  --bootstrap $IMPORT/import-ledger-init.canton
```

This sets up an exportLedger with a set of parties consisting of painters, house owners, and banks along with a handful of paint offer contracts and IOUs.

Define the following helper functions useful to extract parties and contracts via the ledger API:

```
def queryActiveContractsFromDamlLedger(
    hostname: String,
    port: Port,
    tls: Option[TlsClientConfig],
    token: Option[String] = None,
)(implicit consoleEnvironment: ConsoleEnvironment): Seq[CreatedEvent] = {

  // Helper to query the ledger api using the specified command.
  def queryLedgerApi[Svc <: AbstractStub[Svc], Result](
      command: GrpcAdminCommand[_, _, Result]
  ): Either[String, Result] =
    consoleEnvironment.grpcAdminCommandRunner
      .runCommand("sourceLedger", command, ClientConfig(hostname, port, tls),
↪token)
      .toEither

  (for {
    // Identify all the parties on the ledger and narrow down the list to local
↪parties.
    allParties <- queryLedgerApi(
      LedgerApiCommands.PartyManagementService.
↪ListKnownParties(identityProviderId = "")
    )
    localParties = allParties.collect {
      case PartyDetails(party, _, isLocal, _, _) if isLocal => LfPartyId.
↪assertFromString(party)
    }

    // Query the ActiveContractsService for the actual contracts
    acs <- queryLedgerApi(
      LedgerApiCommands.AcsService
        .GetActiveContracts(
          localParties.toSet,
          limit = PositiveInt.MaxValue,
          timeout = NonNegativeDuration.maxTimeout,
        )(consoleEnvironment.environment.scheduler)
    )
  } yield acs.map(_.event)).valueOr(err =>
    throw new IllegalStateException(s"Failed to query parties, ledger id, or acs:
↪$err")
  )
}

def removeCantonSpecifics(acs: Seq[CreatedEvent]): Seq[CreatedEvent] = {
  def stripPartyIdSuffix(suffixedPartyId: String): String =
    suffixedPartyId.split(SafeSimpleString.delimiter).head
```

(continues on next page)

(continued from previous page)

```
  acs.map { event =>
    ValueRemapper.convertEvent(identity, stripPartyIdSuffix)(event)
  }
}

def lookUpPartyId(participant: ParticipantReference, party: String): PartyId =
  participant.parties
    .list(filterParty = party + SafeSimpleString.delimiter)
    .map(_.party)
    .headOption
    .value
```

As the first step, export the active contract set (ACS). To illustrate how to import data from non-Canton ledgers, strip the Canton-specifics by making the party ids generic (stripping the Canton-specific suffix).

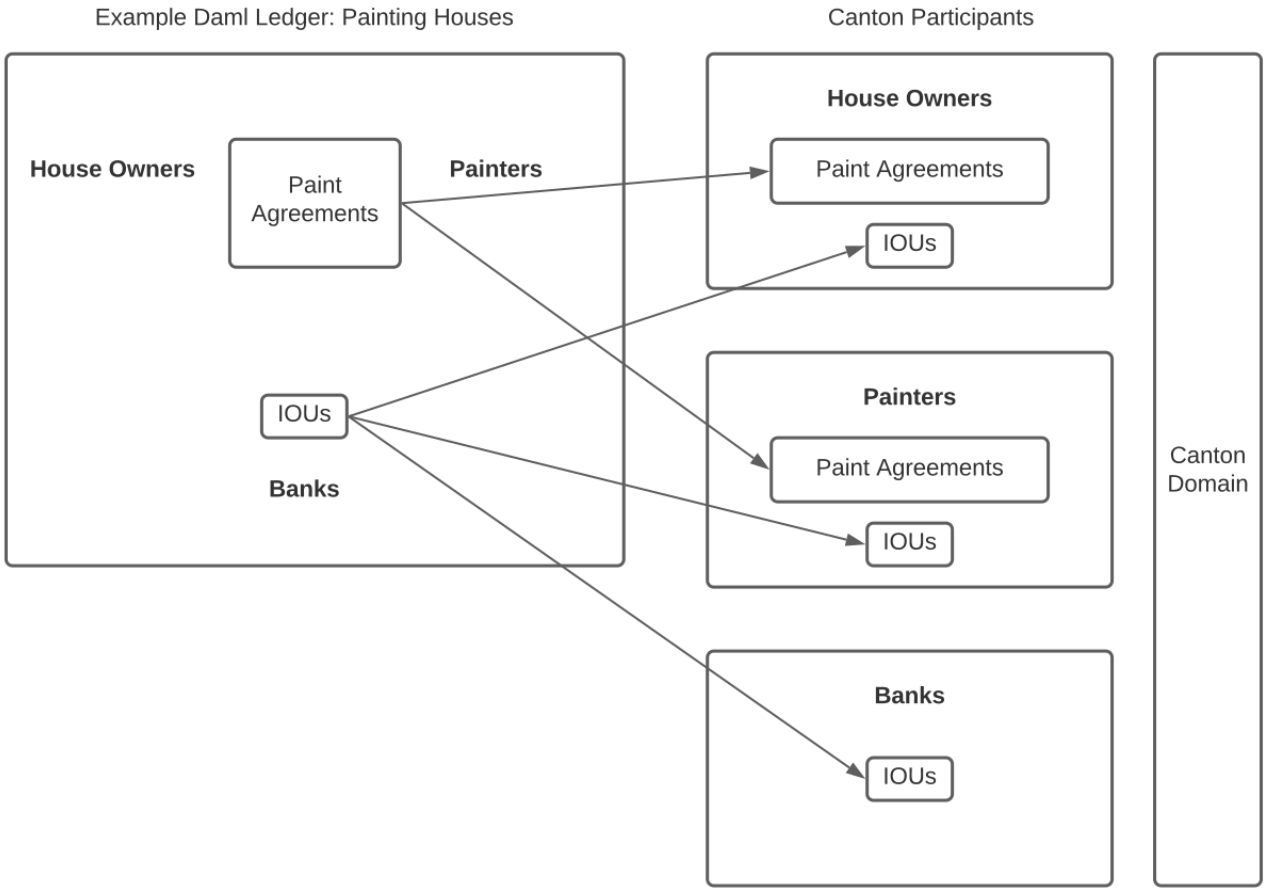
```
val acs =
  queryActiveContractsFromDamlLedger(
    exportLedger.config.ledgerApi.address,
    exportLedger.config.ledgerApi.port,
    exportLedger.config.ledgerApi.tls.map(_.clientConfig),
  )

val acsExported = removeCantonSpecifics(acs).toList
```

Step number two involves preparing the Canton participants and domain by uploading DARs and creating parties. Here we choose to place the house owners, painters, and banks on different participants.

Also modify the events to be based on the newly created party ids.

```
// Decide on which canton participants to host which parties along with their
↪contracts.
// We place house owners, painters, and banks on separate participants.
val participants = Seq(participant1, participant2, participant3)
val partyAssignments =
  Seq(participant1 -> houseOwners, participant2 -> painters, participant3 ->
↪banks)

// Connect to domain prior to uploading dars and parties.
participants.foreach { participant =>
  participant.domains.connect_local(importLedgerDomain)
  participant.dars.upload(darPath)
}

// Create canton party ids and remember mapping of plain to canton party ids.
val toCantonParty: Map[String, String] =
  partyAssignments.flatMap { case (participant, parties) =>
    val partyMappingOnParticipant = parties.map { party =>
      participant.ledger_api.parties.allocate(party, party)
      party -> lookUpPartyId(participant, party).toLf
    }
    partyMappingOnParticipant
  }.toMap
```

(continues on next page)


Example Daml Ledger: Painting Houses
Canton Participants
House Owners
Paint Agreements
Painters
IOUs
Banks
House Owners
Paint Agreements
IOUs
Painters
Paint Agreements
IOUs
Banks
IOUs
Canton Domain

(continued from previous page)

```
// Create traffic on all participants so that the repair commands will pick an
↪identity snapshot that is aware of
// all party allocations
participants.foreach { participant =>
  participant.health.ping(participant, workflowId = importLedgerDomain.name)
}

// Switch the ACS to be based on canton party ids.
val acsToImportToCanton =
  acsExported.map(ValueRemapper.convertEvent(identity, toCantonParty(_)))
```

As the third step, perform the actual import to each participant filtering the contracts based on the location of contract stakeholders and witnesses.

```
// Disconnect from domain temporarily to allow import to be performed.
participants.foreach(_.domains.disconnect(importLedgerDomain.name))

// Pick a ledger create time according to the domain's clock.
val ledgerCreateTime =
  consoleEnvironment.environment.domains
    .getRunning(importLedgerDomain.name)
    .getOrElse(fail("No running domain node"))
    .clock
    .now
    .toInstant

val contractsWithRecomputedContractIds =
  acsToImportToCanton.view
    .map(WrappedCreatedEvent)
    .map { event => utils.contract_data_to_instance(event.toContractData,
↪ledgerCreateTime) }
    .toSeq
    .pipe(recomputeContractIds(participant1.crypto.pureCrypto, _))
    ._1

val createdEventsAndContractsToImport =
  acsToImportToCanton zip contractsWithRecomputedContractIds

// Filter active contracts based on participant parties and upload.
partyAssignments.foreach { case (participant, rawParties) =>
  val parties = rawParties.map(toCantonParty(_))
  val participantAcs = createdEventsAndContractsToImport.collect {
    case (event, contract)
        if event.signatories.intersect(parties).nonEmpty
          || event.observers.intersect(parties).nonEmpty
          || event.witnessParties.intersect(parties).nonEmpty =>
      SerializableContractWithWitnesses(
        contract,
        Set.empty,
      )
  }

  participant.repair.add(importLedgerDomain.name, participantAcs,
↪ignoreAlreadyAdded = false)
}
```

(continues on next page)

(continued from previous page)

```
def verifyActiveContractCounts() = {
  Map[LocalParticipantReference, (Boolean, Boolean)](
    participant1 -> ((true, true)),
    participant2 -> ((true, false)),
    participant3 -> ((false, true)),
  ).foreach { case (participant, (hostsPaintOfferStakeholder,
↪hostsIouStakeholder)) =>
    val expectedCounts =
      (houseOwners.map { houseOwner =>
        houseOwner.toPartyId(participant) ->
          ((if (hostsPaintOfferStakeholder) paintOffersPerHouseOwner else 0)
            + (if (hostsIouStakeholder) 1 else 0))
      }
        ++ painters.map { painter =>
          painter.toPartyId(participant) -> (if (hostsPaintOfferStakeholder)
                                               paintOffersPerPainter
                                             else 0)
        }
        ++ banks.map { bank =>
          bank.toPartyId(participant) -> (if (hostsIouStakeholder) iousPerBank
↪else 0)
        }).toMap[PartyId, Int]

    assertAcsCounts((participant, expectedCounts))
  }
}

/*
  If the test fails because of Errors.MismatchError.NoSharedContracts error, it
↪could be worth to
  extend the scope of the suppressing logger.
 */
loggerFactory.assertLogsUnorderedOptional(
  {
    // Finally reconnect to the domain.
    participants.foreach(_.domains.reconnect(importLedgerDomain.name))
```

To demonstrate that the imported ledger works, let's have each of the house owners accept one of the painters' offer to paint their house.

```
def yesYouMayPaintMyHouse(
    houseOwner: PartyId,
    painter: PartyId,
    participant: ParticipantReference,
): Unit = {
  val iou = participant.ledger_api.acs.await[Iou.Iou](houseOwner, Iou.Iou)
  val bank = iou.value.payer
  val paintProposal = participant.ledger_api.acs
    .await[Paint.OfferToPaintHouseByPainter](
      houseOwner,
      Paint.OfferToPaintHouseByPainter,
      pp => pp.value.painter == painter.toPrim && pp.value.bank == bank,
    )
  val cmd = paintProposal.contractId
```

(continues on next page)

(continued from previous page)

```
    .exerciseAcceptByOwner(iou.contractId)
    .command
  val _ = clue(
    s"$houseOwner accepts paint proposal by $painter financing through ${bank.
↪toString}"
  )(participant.ledger_api.commands.submit(Seq(houseOwner), Seq(cmd)))
}

// Have each house owner accept one of the paint offers to illustrate use of the
↪imported ledger.
houseOwners.zip(painters).foreach { case (houseOwner, painter) =>
  yesYouMayPaintMyHouse(
    lookUpPartyId(participant1, houseOwner),
    lookUpPartyId(participant1, painter),
    participant1,
  )
}

// Illustrate that acceptance of have resulted in
{
  val paintHouseContracts = painters.map { painter =>
    participant2.ledger_api.acs
      .await[Paint.PaintHouse](lookUpPartyId(participant2, painter), Paint.
↪PaintHouse)
  }
  assert(paintHouseContracts.size == 4)
}
```

This guide has demonstrated how to import data from non-Canton Daml Participant Nodes or from a Canton Participant of a lower major version as part of a Canton upgrade.

Repairing Participants

Canton enables interoperability of distributed participants and domains. Particularly in distributed settings without trust assumptions, faults in one part of the system should ideally produce minimal irrecoverable damage to other parts. For example if a domain is irreparably lost, the participants previously connected to that domain need to recover and be empowered to continue their workflows on a new domain.

This guide will illustrate how to replace a lost domain with a new domain providing business continuity to affected participants.

1.33.8.3 Recovering from a Lost Domain

Note: Please note that the given section describes a preview feature, due to the fact that using multiple domains is only a preview feature.

Suppose that a set of participants have been conducting workflows via a domain that runs into trouble. In fact consider that the domain has gotten into such a disastrous state that the domain is

beyond repair, for example:

> The domain has experienced data loss and is unable to be restored from backups or the backups are missing crucial recent history.
> The domain data is found to be corrupt causing participants to lose trust in the domain as a mediator.

Next the participant operators each examine their local state, and upon coordinating conclude that their participants' active contracts are mostly the same . This domain-recovery repair demo illustrates how the participants can

> coordinate to agree on a set of contracts to use moving forward, serving as a new consistent state,
> copying over the agreed-upon set of contracts to a brand new domain,
> fail over to the new domain,
> and finally continue running workflows on the new domain having recovered from the permanent loss of the old domain.

1.33.8.4 Repairing an actual Topology

To follow along with this guide, ensure you have *installed and unpacked the Canton release bundle* and run the following commands from the canton-X.Y.Z directory to set up the initial topology.

```
export CANTON=`pwd`
export CONF="$CANTON/examples/03-advanced-configuration"
export REPAIR="$CANTON/examples/07-repair"
bin/canton \
  -c $REPAIR/participant1.conf,$REPAIR/participant2.conf,$REPAIR/domain-repair-
↪lost.conf,$REPAIR/domain-repair-new.conf \
  -c $CONF/storage/h2.conf,$REPAIR/enable-preview-commands.conf \
  --bootstrap $REPAIR/domain-repair-init.canton
```

To simplify the demonstration, this not only sets up the starting topology of

> two participants, participant1 and participant2 , along with
> one domain lostDomain that is about to become permanently unavailable leaving participant1 and participant2 unable to continue executing workflows,

but also already includes the ingredients needed to recover:

> The setup includes newDomain that we will rely on as a replacement domain, and
> we already enable the enable-preview-commands configuration needed to make available the repair.change_domain command.

In practice you would only add the new domain once you have the need to recover from domain loss and also only then enable the repair commands.

We simulate lostDomain permanently disappearing by stopping the domain and never bringing it up again to emphasize the point that the participants no longer have access to any state from domain1. We also disconnect participant1 and participant2 from lostDomain to reflect that the participants have given up on the domain and recognize the need for a replacement for business continuity. The fact that we disconnect the participants at the same time is somewhat artificial as in practice the participants might have lost connectivity to the domain at different times (more on reconciling contracts below).

```
lostDomain.stop()
Seq(participant1, participant2).foreach { p =>
  p.domains.disconnect(lostDomain.name)
  // Also let the participant know not to attempt to reconnect to lostDomain
  p.domains.modify(lostDomain.name, _.copy(manualConnect = true))
}
```

Even though the domain is the node that has broken , recovering entails repairing the participants using the newDomain already set up. As of now, participant repairs have to be performed in an offline fashion requiring participants being repaired to be disconnected from the the new domain. However we temporarily connect to the domain, to let the topology state initialize, and disconnect only once the parties can be used on the new domain.

```
Seq(participant1, participant2).foreach(_.domains.connect_local(newDomain))

// Wait for topology state to appear before disconnecting again.
clue("newDomain initialization timed out") {
  eventually()(
    (
      participant1.domains.active(newDomain.name),
      participant2.domains.active(newDomain.name),
    ) shouldBe (true, true)
  )
}

// Run a few transactions on the new domain so that the topology state chosen by
↪the repair commands
// really is the active one that we've seen
participant1.health.ping(participant2, workflowId = newDomain.name)

Seq(participant1, participant2).foreach(_.domains.disconnect(newDomain.name))
```

With the participants connected neither to lostDomain nor newDomain , each participant can

locally look up the active contracts assigned to the lost domain using the testing.pcs_search command made available via the features.enable-testing-commands configuration, and invoke repair.change_domain (enabled via the features.enable-preview-commands configuration) in order to move the contracts to the new domain.

```
// Extract participant contracts from "lostDomain".
val contracts1 =
  participant1.testing.pcs_search(lostDomain.name, filterTemplate = "^Iou",
↪activeSet = true)
val contracts2 =
  participant2.testing.pcs_search(lostDomain.name, filterTemplate = "^Iou",
↪activeSet = true)

// Ensure that shared contracts match.
val Seq(sharedContracts1, sharedContracts2) = Seq(contracts1, contracts2).map(
```

(continues on next page)

(continued from previous page)

```
  _.filter { case (_isActive, contract) =>
    contract.metadata.stakeholders.contains(Alice.toLf) &&
    contract.metadata.stakeholders.contains(Bob.toLf)
  }.toSet
)

clue("checking if contracts match") {
  sharedContracts1 shouldBe sharedContracts2
}

// Finally change the contracts from "lostDomain" to "newDomain"
participant1.repair.change_domain(
  contracts1.map(_._2.contractId),
  lostDomain.name,
  newDomain.name,
)
participant2.repair.change_domain(
  contracts2.map(_._2.contractId),
  lostDomain.name,
  newDomain.name,
  skipInactive = false,
)
```

Note: The code snippet above includes a check that the contracts shared among the participants match (as determined by each participant, sharedContracts1 by participant1 and sharedContracts2 by participant2). Should the contracts not match (as could happen if the participants had lost connectivity to the domain at different times), this check fails soliciting the participant operators to reach an agreement on the set of contracts. The agreed-upon set of active contracts may for example be

> the intersection of the active contracts among the participants
> or perhaps the union (for which the operators can use the repair.add command to create the contracts missing from one participant).

Also note that both the repair commands and the testing.pcs_search command are currently preview features, and therefore their names may change.

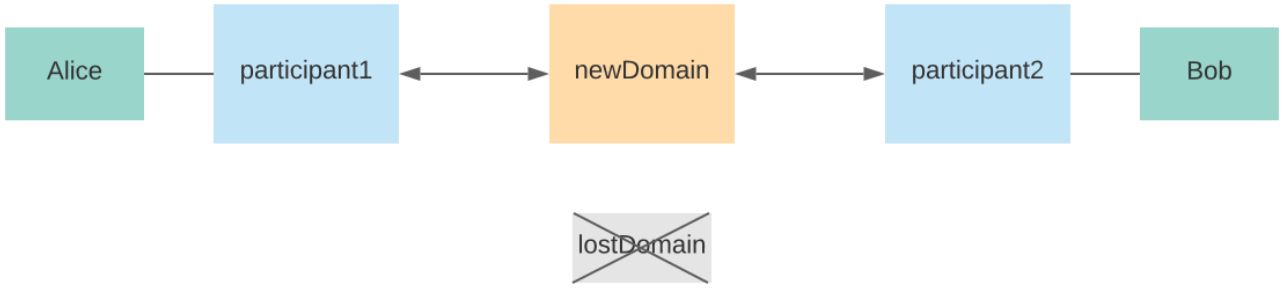
Once each participant has associated the contracts with newDomain, let's have them reconnect, and we should be able to confirm that the new domain is able to execute workflows from where the lost domain disappeared.

```
Seq(participant1, participant2).foreach(_.domains.reconnect(newDomain.name))

// Look up a couple of contracts moved from lostDomain
val Seq(iouAlice, iouBob) = Seq(participant1 -> Alice, participant2 -> Bob).map {
  case (participant, party) =>
    participant.ledger_api.acs.await[Iou.Iou](party, Iou.Iou, _.value.owner ==
↪party.toPrim)
}

// Ensure that we can create new contracts
Seq(participant1 -> ((Alice, Bob)), participant2 -> ((Bob, Alice))).foreach {
  case (participant, (payer, owner)) =>
```

(continues on next page)

(continued from previous page)

```
    participant.ledger_api.commands.submit_flat(
      Seq(payer),
      Seq(
        Iou
          .Iou(
            payer.toPrim,
            owner.toPrim,
            Iou.Amount(value = 200, currency = "USD"),
            List.empty,
          )
          .create
          .command
      ),
    )
}

// Even better: Confirm that we can exercise choices on the moved contracts
Seq(participant2 -> ((Bob, iouBob)), participant1 -> ((Alice, iouAlice))).foreach
↪{
  case (participant, (owner, iou)) =>
    participant.ledger_api.commands
      .submit_flat(Seq(owner), Seq(iou.contractId.exerciseCall().command))
}
```



In practice, we would now be in a position to remove the lostDomain from both participants and to disable the repair commands again to prevent accidental use of these dangerously powerful tools.

This guide has demonstrated how participants can recover from losing a domain that has been permanently lost or somehow become irreparably corrupted.

Repair Macros

Some operations are combined as macros, which are a series of consecutive repair commands, coded as a single command. While we discourage you from using these commands on your own, we document them here for the sake of completeness. These macros are available only in the enterprise edition.

1.33.8.5 Clone Identity

Many nodes can be rehydrated from a domain, as long as the domain is not pruned. In such situations, you might want to reset your node while keeping the identity and the secret keys of the node. This can be done using the repair macros.

You need local console access to the node. If you are running your production node in a container, you need to create a new configuration file that allows you to access the database of the node from an interactive console. Make sure that the normal node process is stopped and that nothing else is accessing the same database (e.g. ensure that replication is turned on). Also, make sure that the nodes are configured to not perform auto-initialization, as this would create a new identity. You ensure that by setting the corresponding auto-init configuration option to false:

```
canton.participants.myparticipant.init.auto-init = false
```

Then start Canton interactively using:

```
./bin/canton -c myconfig --manual-start
```

Starting with `--manual-start` will prevent the participant to attempt to reconnect to the domains. Then, you can download the identity state of the node to a directory on the machine you are running the process:

```
repair.identity.download(participant, tempDirParticipant)
repair.dars.download(participant, tempDirParticipant)
participant.stop()
```

This will store the secret keys, the topology state and the identity onto the disk in the given directory. You can run the `identity.download` command on all nodes. However, mediator and sequencer nodes will only store their keys in files, as the sequencer's identity is attached to the domain identity and the mediator's identity is set only later during initialization.

The `dars.download` command is a convenience command to download all dars that have been added to the participant via the console command `participant.dars.upload`. Dars that were uploaded through the Ledger API need to be manually re-uploaded to the new participant.

Once the data is stored, stop the node and then truncate the database (please back it up before). Then restart the node and upload the identity data again:

```
participant.start()
repair.identity.upload(participant, tempDirParticipant)
repair.dars.upload(participant, tempDirParticipant)
```

Please note that dar uploading is only necessary for participants.

Now, depending on the node type, you need to re-integrate the node into the domain. For the domain nodes, you need to grab the static domain parameters and the domain id from the domain manager.

If you have remote access to the domain manager, you can run

```
val domainId = domainManager1.id
val domainParameters = domainManager1.service.get_static_domain_parameters
```

You also want to grab the mediator identities for each mediator using:

```
val mediatorId = mediator.id
```

For the sequencer, rehydration works only if the domain uses a blockchain; the database-only sequencers cannot rehydrate. So rehydration for blockchain-based sequencers will be:

```
repair.identity.upload(newSequencer, tempDirSequencer)
newSequencer.initialization.initialize_from_beginning(domainId, domainParameters)
newSequencer.health.wait_for_initialized()
```

For the domain manager, it looks like:

```
repair.identity.upload(domainManager2, tempDirDomainManager)
domainManager2.setup.init(
  SequencerConnections.single(newSequencer.sequencerConnection)
)
domainManager2.health.wait_for_initialized()
```

For the mediator, it would be:

```
repair.identity.upload(mediator, tempDirMediator)
mediator.mediator.initialize(
  domainId,
  mediatorId,
  domainParameters,
  SequencerConnections.single(newSequencer.sequencerConnection),
  topologySnapshot = None,
)
mediator.health.wait_for_initialized()
```

For a participant, you would reconnect it to the domain using a normal connect:

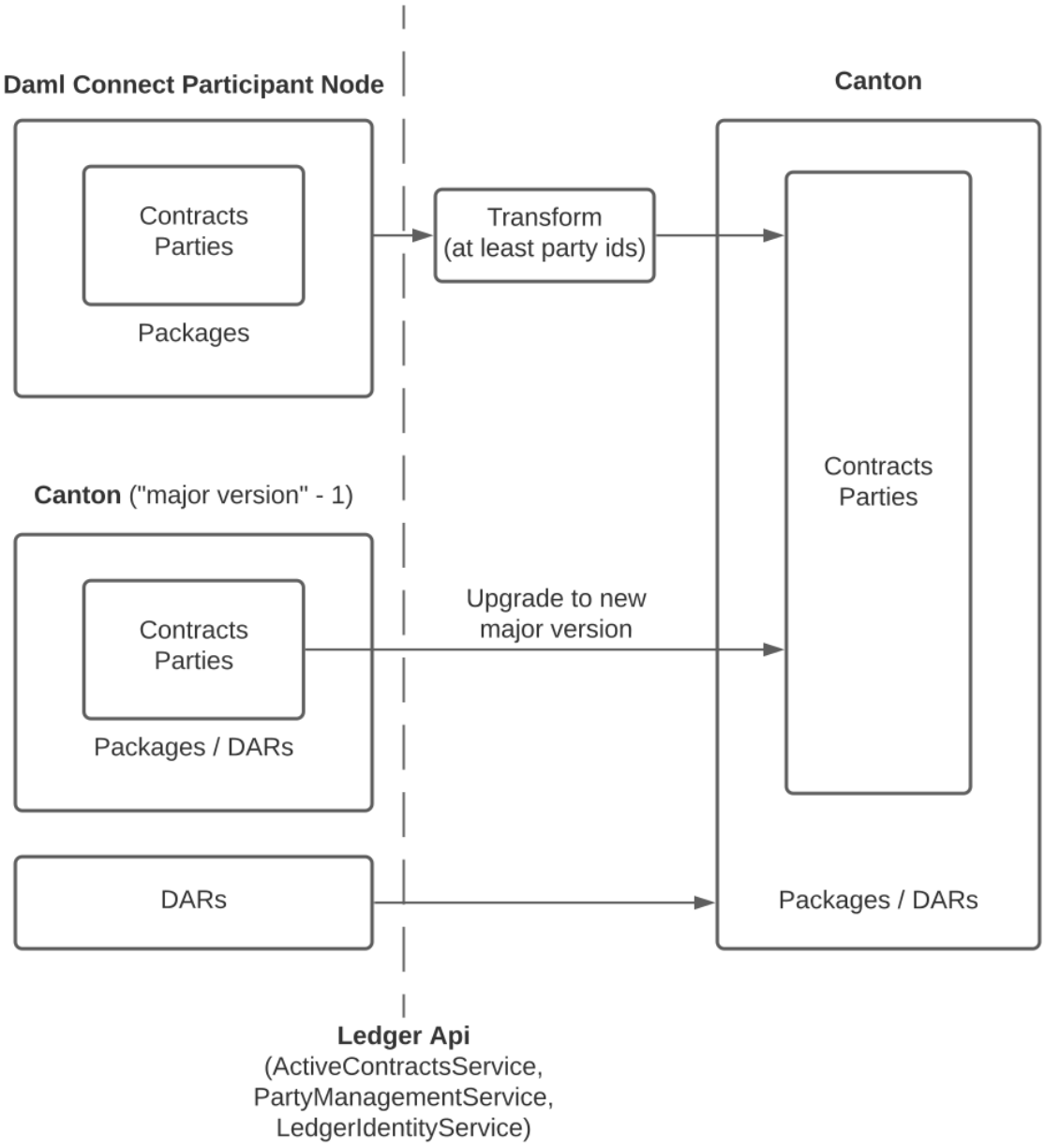
```
participant.domains.connect_local(sequencer)
```

Note that this will replay all transactions from the domain. However, command deduplication will only be fully functional once the participant catches up with the domain. Therefore, you need to ensure that applications relying on command deduplication do not submit commands during recovery.

1.33.8.6 Importing existing Contracts

You may have existing contracts, parties, and DARs in other Daml Participant Nodes (such as the Daml sandbox) that you want to import into your Canton-based participant node. To address this need, you can extract contracts and associated parties via the ledger API, modify contracts, parties, and daml archived as needed, and upload the data to Canton using the *Canton Console*.

You can also import existing contracts from Canton as that is useful as part of Canton upgrades across major versions with incompatible internal storage.


Daml Connect Participant Node
Canton
Contracts
Parties
Packages
Transform
(at least party ids)
Contracts
Parties
Canton ("major version" - 1)
Contracts
Parties
Packages / DARs
Upgrade to new
major version
DARs
Packages / DARs
Ledger Api
(ActiveContractsService,
PartyManagementService,
LedgerIdentityService)

1.34 Troubleshooting Guide

1.34.1 Introduction

Distributed systems can fail in many ways and finding the cause of an error is not straight forward. This guide here captures the common steps our engineers take when trying to troubleshoot issues found during development or support.

1.34.2 Enable Information Gathering

The following switches / steps should be taken in order to improve analyzing errors. Without these, you might not be able to diagnose harder issues.

Create Health Dumps
Ensure that you are able to create *Health Dumps*. You need to share these health dumps during any support request. The health dumps provide a lot of diagnostic information that we need to troubleshoot issues.

Turn on Debug Logging
Turn on debug logging by starting the process with:

```
./bin/canton -v
```

or even:

```
./bin/canton -debug
```

The `-v` flag will turn on debug logging of all `com.digitalasset.canton` loggers, whereas `--debug` will turn on debug logging of all libraries too. Please also see *Logging*.

Turn on Detailed API Logging
You might want to turn on *detailed api logging*. This will write all incoming commands and the outgoing data into the log file and it will allow you to inspect the details of a command that leads to failures. Beware that if your commands contain sensitive data, this data will then be written to the log file.

Turn on metrics collection
See *Canton Metrics <canton-metrics>*. If you don't have a metrics system, you can report metrics to CSV files and JMX beans by configuring:

```
metrics.report-jvm-metrics = true

canton.monitoring.metrics.reporters = [{
  type = csv
  directory = "metrics"
  interval = 5s
  filters = [{
    contains = "canton.updates-published"
  }, {
    contains = "sequencer-client.event-handle"
  }, {
    contains = "sequencer.processed"
  }, {
    contains = "executor.queued"
  }, {
```

(continues on next page)

(continued from previous page)

```
    contains = "executor.running"
  }, {
    contains = "executor.waittime"
  }, {
    contains = "jvm.memory_usage.heap"
  }, {
    contains = "jvm.memory_usage.non-heap"
  }, {
    contains = "jvm.thread_states"
  }]
}, {
  type = jmx
}]
```

This will periodically write selected metrics to CSV files (one file per metric). It will also expose all available metrics as JMX beans; therefore, you can use VisualVM to look at metric values.

The CSV reporter needs to have a `filters` parameter, because otherwise Canton will report all available metrics, which would substantially slow-down Canton. The JMX reporter does not need a `filters` parameter, because JMX beans only get evaluated when you actually look at them using VisualVM. So the JMX reporter is great for exploring different metrics initially. The CSV reporter is preferable, if you want to record metrics without human intervention.

Turn on database query cost monitoring

Enable `canton.monitoring.log-query-cost.every = 60s`. This will capture query cost statistics and might help diagnose latency / indexing issues with your database, as explained in *How to Diagnose Slow Database Queries*.

Turn on slow futures supervision

Enable `canton.monitoring.log-slow-futures = yes` which will track some operations and alert if they are taking too long (disabled by default to reduce the overhead).

Do not disable deadlock detection (enabled by default)

Deadlock detection (`canton.monitoring.deadlock-detection`) will periodically test if the JVM executes new tasks in a timely manner. It will log the following warning, if this is not the case: *Task runner <name> is stuck or overloaded for 5s* . Failure of this check may indicate that the CPU is overloaded, the execution context is too small. Usually the check resolves itself with a subsequent log message: *Task runner <name> is just overloaded, but operating correctly. Task got executed in the meantime.* If this message does not appear, all available threads are blocked for some reason. Their stack-traces will be logged additionally. However, all threads being blocked are not common. They should not happen. Therefore, normally this check just indicates that your system is overloaded.

Configure delay logging

Delay logging (`canton.monitoring.delay-logging-threshold`, default `20s`) will log a warning, if a node falls behind with processing messages from the sequencer. Such a warning indicates that the node is overloaded. As a rule of thumb, configure the maximum latency, i.e., the maximum time it should take Canton to process a command.

Do not disable trace context propagation (enabled by default)

Every request will receive a unique trace id. The trace id is included in log messages referring to that request. If trace context propagation is enabled (`canton.monitoring.tracing.propagation = enabled`), different nodes will use the

same trace id for a request. This makes it easier for you to identify log messages across different nodes that refer to the same request.

1.34.3 Key Knowledge

Canton Transaction Processing Steps

Canton transaction processing has the following key steps involved. When we debug, we obviously try to find out which of the steps fails / is slow / faulty. This can help you to narrow down the component and the issue. As all the message exchange happens via the sequencer, you effectively observe whether the information came into the node and where the action that the node was supposed to take was taken by responding with a message to the sequencer (or emitting a command result on the ledger API). The phases are:

- Phase 1: Submitting participant prepares the confirmation request based on the Daml command input . The confirmation request is sent to the sequencer, addressing the mediator and the validating participants.
- Phase 2: The mediator receives the request from the sequencer, registers the transaction and starts to wait for confirmations.
- Phase 3: The validating participants receive the confirmation request from the sequencer and perform their validations. The two main checks that happen here are: validation (is the transaction correct and properly authorized?) & conflict detection (are all contracts that are spent or fetched in the transaction still active?).
- Phase 4: The confirming participants, a subset of the validating participants, send their verdict on each sub-transaction they are privy via the sequencer to the mediator. The verdict can be `LocalApprove` or some rejection reason.
- Phase 5: The mediator receives the mediator responses (approvals and rejections) from the participants via the sequencer and validates them. If the mediator receives enough responses for the given transaction, it will compute the Verdict , which is the final decision on the transaction.
- Phase 6: The mediator sends its verdict to all validating participants of a transaction via the sequencer.
- Phase 7: The participants receive the mediator verdict and register it to the record order publisher. While the validation can happen in parallel, the record publisher will ensure that the transactions are emitted in order.

For each phase a log line that should appear at the beginning and one that appears at the end of the phase.

Internal Errors

If internal consistency checks fail and indicate a possible bug in Canton, Canton will include the term `internal error` into the log message. Please contact support, if you see an internal error.

Canton Error Codes

All non-internal warnings and errors are *logged consistently* (or at least we aspire to do).The error code information listed in the documentation should contain all information you need in order to understand and possibly resolve the issue.

1.34.4 Log Files

Canton Trace Ids

All Canton log statements contain a *trace-id*. This tracing is turned on by default and the `trace-id` is passed between the distributed processes:

```
c.d.c.p.p.s.InFlightSubmissionTracker:participant=participant1
tid:d5df95972a95b5ff00cb5cc3346c545f - NOT_SEQUENCED_TIMEOUT(2,
↪d5df9597):
Transaction was not sequenced within the pre-defined max sequencing
↪time and has
therefore timed out err-context:{location=SubmissionTrackingData.
↪scala:175,
timestamp=2022-10-19T17:45:56.393151Z}
```

In above example, we see the trace id twice: `tid:d5df95972a95b5ff00cb5cc3346c545f` and `NOT_SEQUENCED_TIMEOUT(2,d5df9597)`. By filtering according to the `trace-id`, you can find almost all log statements that relate to a particular command. However, sometimes, we also need to find out the command id of a transaction. You can do that by grepping for the rosetta stone , which is one particular log line that contains both strings:

```
2023-07-04 12:03:26,517 [] INFO
c.d.c.p.a.s.c.CommandSubmissionServiceImpl:participant=participant1
tid:35e389f0e41fd0273443dd866ff9e347 - Submitting commands for
↪interpretation,
commands -> {readAs: [], deduplicationPeriod: {duration: 'PT168H'},
submittedAt: '2023-07-04T10:03:26.514885Z', ledgerId: 'participant1',
applicationId: 'CSsubmitAndWaitBasic',
submissionId: 'CSsubmitAndWaitBasic-alpha-410b4d7b1b585-submission-0',
actAs: ['CSsubmitAndWaitBasic-alpha-410b4d7b1b585-party-
↪0::122035bd93d74879ce582adf5aa04a809b4b20618d39c1a9c2a17d35c29ab1ed098f
↪'],
commandId: 'CSsubmitAndWaitBasic-alpha-410b4d7b1b585-command-0',
workflowId: 'CSsubmitAndWaitBasic-410b4d7b1b585'}.
```

The first string is again the trace id. Additionally, the commandId of the transaction, the applicationId, the submissionId and the workflowId are logged and can be used to filter the logs.

Extract the Context of a Log Message

The log lines often also contain the context of the component. Examples:

- This log line tells us which component of which participant (participant1) of which domain connection (da) has been emitting this log line. It also includes the trace id of the underlying request:

  ```
  2022-10-04 15:55:50,077 [] DEBUG
  c.d.c.p.p.TransactionProcessingSteps:participant=participant1/
  ↪domain=da
  tid:461cae6245cfaadc87c2481a17d7e1bb - Preparing batch for
  ↪transaction
  submission
  ```

- During tests, the log line includes the name of the test. In this case, it is `SimplestPingIntegrationTestInMemory`:

  ```
  :: 2022-10-04 15:55:50,077 [] DEBUG c.d.c.p.p.
  ↪TransactionProcessingSteps:SimplestPingIntegrationTestInMemory/
  ```

(continues on next page)

(continued from previous page)

```
participant=participant1/domain=da
 ↪tid:461cae6245cfaadc87c2481a17d7e1bb
- Preparing batch for transaction submission
```

Compare with a Happy Path Successful Logging Trace

Many components will log something and it is impossible to document every micro-step that happens (as this is also subject to change). But it makes sense to compare a failure trace with a successful transaction trace. To get such a trace, you start up a canton simple topology example setup and run a simple:

```
participant1.health.ping(participant2)
```

You then open the log file and filter for the command processing of that ping (search for Starting ping). This will give you a clean happy path trace . You can then subsequently compare your failure trace to the happy-path trace and look for the differences, i.e. where did the steps start to take a different path etc.

Use the API Request Logger to Locate the Component

One key logging component is the `ApiRequestLogger`. This component is injected into the gRPC library and will log every incoming and outgoing request / message. Therefore, we can easily observe when a transaction left a node and when it arrived at a subsequent node. If api logging is turned on, the api request logger will print the full detail of all the gRPC messages into the log files.

1.34.5 Using LNAV to View Log Files

Setup and Use LNAV

Setup lnav for viewing logs as described in *viewing logs*. It will require a few minutes to get used to it, but the payoff of this investment is great and comes fast. **In particular get familiar with loading multiple files, filtering, searching and jumping to errors**.

Open Multiple Log Files in one LNAV Session

Generally, when you start reading log files, then open the log files of all involved nodes in a single lnav session (if the files are small enough): `lnav participant1.log domain.log participant2.log`

Split Log Files if they are too big

If your log files are too big the unix utility `split` can be used to split the file into chunks.

Uncompress GZ Log files for faster reading

Normally, log files are compressed when you get them. Lnav works much better and faster if you pass uncompressed files on the command line.

Easily Navigate to the First Logged Error

Then hit `g` to go to the beginning of the file and subsequently `w` or `e` to get to the first warning or error. Usually, the first error gives you the hint on what is going on.

Look at All Warnings and Errors

Canton's error reporting has been designed to log a warning/error whenever it detects that something is not working as it should. Therefore, any problem will likely show up in the log file. On the flip side, Canton may log a huge number of warnings/errors, in particular if a node or the database goes down. If the first warning or error does not completely explain the situation, it is important to look at all such messages. Use the following recipe:

1. Set the minimum log level to WARN to display only warnings and errors (`:set-min-log-level warn`).

2. Look at the first message. Mark the message (pressing `m`) so you can later get back to the message.
3. Define an out-filter to hide the first message and all similar messages.
4. Repeat steps (2) and (3) until you have filtered out all messages.
5. Disable all out-filters. You can now press u and U to step through all marked warning and error messages.

Filter Irrelevant Items

One useful strategy when working with logs is to continuously remove lines that are not relevant, adding filter-out until only the relevant log messages remain.

Show Gap In Logging Times

Once you start filtering for a particular command trace, you might want to hit shift-t . This will show you the delta time between the first log line and the subsequent one. Usually, you just need to find the gap . This will tell you immediately where something got stuck / slow / timed out:

- open the log files of all components
- search for the first error / warn (i.e. hit `w` or `e`)
- pick the trace-id (as described above) and filter for it
- hit `shift-t` and find the gap.

1.34.6 Setup Issues

Connect to each node and check the status: *<node>* `.health.status`
Are the nodes up and running?
Are the nodes *connected to a sequencer*? Errors that often happen here are:

- public-apis / ledger-api addresses are not set to `0.0.0.0` and are still binding to `localhost` (default value for security reasons).
- you are using TLS on the server side, but on the client side you have defined the URL as `http://`.
- the chosen port is not correct.

If you are running into TLS connectivity issues, turn on `-debug` and check the detailed netty logs for hints. These libraries tend to log necessary information only on debug level. You can also increase the debugging information level by starting canton with *-Djavax.net.debug=all*.
Try to confirm that your setup works by running a ping:

```
participant1.health.ping(participant2)
```

1.34.7 Timeout Errors

Any transaction that is submitted to Canton will either be successfully worked off (accepted or rejected), or eventually timeout. If a transaction hits a timeout, the application will be informed by an appropriate completion event on the Ledger API about the rejection reasons. We can hit the following timeouts in Canton (you can get further timeouts from the command service):

```
NOT_SEQUENCED_TIMEOUT
LOCAL_VERDICT_TIMEOUT
MEDIATOR_SAYS_TX_TIMED_OUT
LOCAL_VERDICT_LEDGER_TIME_OUT_OF_BOUND
LOCAL_VERDICT_SUBMISSION_TIME_OUT_OF_BOUND
```

Such a timeout usually means that some component is either:

offline - resolve by checking that all nodes are healthy (`health.status()`) and are connected with each other.
overloaded - resolve by tuning according to our *performance configuration guide*.
unable to complete the transaction processing within the given time (i.e. transactions are too big) - resolve by increasing the timeouts as described in our *performance configuration guide*.

Use a ping to determine if your system is broken or just slow / overloaded / contentious
Many issues only surface under high load. Therefore, it often makes sense to diagnose timeout issues using a:

```
participant1.health.ping(...)
```

while the system is idle. If the ping works, then you have likely a throughput / performance / contention issue and you should use one of the other guides to continue debugging.
If the ping doesn't work and never did before, you should check the setup troubleshooting guide.
If previously, transaction processing worked and now stopped working, while all nodes are up and running, and reporting to be healthy, you should raise an issue with support.
By turning on diagnostics information collection as explained above, you can then figure out which step of transaction processing failed by comparing the trace in the logs to the Phase 1-7 explanation, isolating out which component did not respond.

1.34.8 Auth Errors

For security reasons, Canton removes all details from auth errors. On the client side, you usually only see `PERMISSION_DENIED/An error occurred. Please contact the operator and inquire about the request <no-correlation-id>`, so you need to inspect server logs to debug auth errors.

To use an auth-enabled ledger API, the caller needs to attach an access token to the gRPC request. These tokens are attached in the `Authorization` HTTP header. To see headers attached to incoming and outgoing requests, you need to set the log level to `TRACE`. `ApiRequestLogger` will then output log lines containing `received headers` or `sending response headers`.

Filter-in expressions for lnav:

com.digitalasset.canton.ledger.api.auth.Authorizer
c.d.l.a.a.i.AuthorizationInterceptor
c.d.c.n.g.ApiRequestLogger

Common patterns from the canton log:

`PERMISSION_DENIED(7,0): Could not resolve is_deactivated status for user`
You are using a token for a user that is not (yet) allocated. The log line contains the name of the user that needs to be allocated.

`PERMISSION_DENIED(7,0): Claims are only valid for applicationId`
You are using a wrong application Id when submitting commands. The log line contains the expected application ID. Note that the application ID must be equal to the daml user name when using ledger API access tokens.

`UNAUTHENTICATED(6,0): The command is missing a (valid) JWT token`

You did not attach a token to the request, or the token could not be decoded. Use JWT.IO to verify that the token string is a valid JWT.

`PERMISSION_DENIED(7,0): Claims do not authorize to act as party`
: The log line contains the name of the missing claim, but not the actual claims. When using tokens based on user names (Audience Based Tokens and Scope Based Tokens), consult the user management service to see whether you need to grant more rights to the user. When using tokens based on party names (Custom Claims Access Tokens), debug the token in JWT.IO.

1.34.9 Performance Issues

How to obtain a performant system is *extensively documented*.

If you have followed that documentation, we can assume that:

Your database pools are sufficiently sized: check metric `db-storage.queue`.
You have set the right settings with respect to:
- number of threads (check cpu usage)
- number of database connections (connection pool size) (`max-connections` in storage)
- high-throughput sequencer settings (`sequencer.writer.type = high-throughput`)

The database server is using SSDs and not spinning disks, and the latency to the database is low.
The database has enough memory to keep the indexes in memory (shared_buffers!) and is properly configured.
The number of connections to the database aligns with the available resources on the database. A database can not concurrently serve more than one request per CPU. Allocating too many connections will lead to contention and slow down the database (latency under load goes up as you queue on the db).
You are not using one of the slow DLT layers such as Fabric or Besu that are simply limited in their throughput (`sequencer.type = database`).
You have enough spare CPU capacity (cpu usage is not at 100%).
You don't have other systems competing for resources.
The max inflight transaction resource limits on the participant (`participant1.resources.set_resource_limits`) have been set carefully. The resource limits are low enough so that an application cannot overload Canton. The resource limits are high enough such that applications can submit commands at the desired target rate.
You are able to load the system fully. I.e. the load generator that you apply is submitting faster than the system can handle (i.e. you throttle using, for example, max 1000 pending commands, the latency grows linearly with num pending commands).

If you have done all that, you might have reached the limit of what the Canton version you are using can do. The next step is then to find out which component is creating the bottleneck. Generally, it is either one of the nodes or the database.

1.34.9.1 How to Measure Database Performance

To get a first impression of database performance, enable the following metrics:

Metrics containing `executor.waittime`. These metrics show the time (in millis) a db command needs to wait until Canton sends it to the db. High values indicate that the db is a bottleneck.
Metrics containing `executor.queued`. These metrics show the number of db commands waiting in a queue for being sent to the db. High values indicate that the db is a bottleneck.
Metrics containing `executor.running`. These metrics show the number of tasks currently being executed by the db. Very high values indicate that Canton is overloading the db. Very low values indicate that Canton is not fully loading the db. The number of db connections can be configured via `canton.<path-to-my-node>.storage.parameters.max-connections`.

1.34.9.2 How to Diagnose Slow Database Queries

If database metrics indicate that the database is a bottleneck you may want to obtain more detailed information on query performance. For that, you need to enable query cost monitoring (`canton.monitoring.log-query-cost.every = 60s`). Once you have done that, Canton will log every 60 seconds a report on query statistics:

```
2022-08-16 07:12:35,528 [slick-diexec_domain-4-7] INFO  c.d.c.r.
↪DbStorage:domain=diexec_domain - Here is our list of the 15 most expensive
↪database queries for canton.db-storage.general.executor:
count=    598 mean=  13.61 ms total=  8.1 s com.digitalasset.canton.domain.
↪sequencing.sequencer.store.DbSequencerStore.saveWatermark(DbSequencerStore.
↪scala:593)
count=    598 mean=   8.82 ms total=  5.3 s com.digitalasset.canton.domain.
↪sequencing.sequencer.store.DbSequencerStore.fetchWatermark(DbSequencerStore.
↪scala:621)
count=      1 mean=  29.48 ms total=  0.0 s com.digitalasset.canton.domain.
↪sequencing.authentication.DbMemberAuthenticationStore.
↪expireNoncesAndTokens(MemberAuthenticationStore.scala:234)
count=      2 mean=   9.37 ms total=  0.0 s com.digitalasset.canton.topology.
↪store.db.DbTopologyStore.$anonfun$queryForTransactions$2(DbTopologyStore.
↪scala:387)
count=      1 mean=  18.52 ms total=  0.0 s
```

The information in here can be very useful:

`count` means how often has this query run in the last period.
`mean` means what was the average execution time of that query
`total` = `count` * `mean`
`saveWatermark(DbSequencerStore.scala:593)` is really the query with the place in the source code that is being run

Please note that the execution time of the query does not include queuing time in the connection pool. The time is really the time it took from sending to the JDBC driver to getting the result back.

Now, you do the following analysis:

if you have for example `max-connections = 4` and you log once a minute, if the total time of the queries approaches 240s, then you are obviously using up all db connections that are available.

if a single query runs for `60s`, then that query might be a sequential bottleneck, as it has been running for 60s out of the 60s interval.
the mean time should also tell you roughly the db latency, as there are some cheap read queries that should run *< 1ms*. If these queries take a long time, then you know that the database has high latencies or is overloaded.

1.34.9.3 How to find the Bottleneck

In some situations, you would like to understand which component is causing a particular bottleneck. You can do that using the following technique.

Theory

In a model system with several computing stages:

Input -> Stage1 -> Stage2 -> Stage3 -> Stage4 -> Stage5 -> Output

The maximum throughput of the system is given by the minimum of the maximum throughputs of all stages. Let's assume that the max throughput is limited by Stage3 that has 100 tx/s.

Now, if you have an input source that will throttle its submission based on the number of open requests , then we know that the average latency of each transaction is going to be

latency = num-open-requests / max-throughput

The latency will grow linearly with the number of open requests. Now, as we previously defined that

throughput(Stage 3) < throughput (all other Stages)

We know that the open requests will be starting to pile up in front of Stage 3, because all other stages are processing every transaction much faster.

Therefore, if we run the system under full load with N pending requests, such that the observed latency is large compared to the zero load latency of the system, then the bottleneck is trivially observable from the trace of a command: there will be a gap in the trace of a command, where the transaction is not being processed for (*observed latency - zero load latency*). That gap is the sequential bottleneck.

Practical

1. Find out what the zero load latency of your system is by running a simple ping over an idle system. A ping does three end-to-end Daml transactions, so your zero load latency is just a third of the observed ping latency.
2. Run the system under full load again, including debug logging. You should be able to load the system such that the observed latency is at least an order of magnitude larger than the zero load latency.
3. Open the log files and pick a transaction in the middle of your test run:
   Look for TransactionAccepted somewhere in the log file and pick the trace-id
   Filter for the trace-id and find the command-id. Add the command-id to the filter
   Hit Shift-T to see the time differences.
   Find the gaps

To increase confidence, repeat this assessment on a few more transactions.

1.34.10 Contention

1.34.10.1 Why do you get contention

This section here explains you how to deal with situations where many commands are failing with errors such as:

```
LOCAL_VERDICT_LOCKED_CONTRACTS
LOCAL_VERDICT_LOCKED_KEYS
LOCAL_VERDICT_INCONSISTENT_KEY
LOCAL_VERDICT_INACTIVE_CONTRACTS
LOCAL_VERDICT_DUPLICATE_KEY
CONTRACT_NOT_FOUND
DUPLICATE_CONTRACT_KEY
```

Canton is not just a distributed system, but a distributed **racy** system where different independent actors may race for contracts or other resources. As a simple example: if you have an offer contract that can be accepted by a buyer and revoked by the seller, then the decision of the buyer to accept can race with a decision of the seller to revoke the offer.

Now, a distributed decision system with individual actors can be **accidentally racy** or **intentionally racy**. Let's explain the difference between the two:

> **Intentionally racy**: You are putting out an offer for anyone interested on a first-come first serve basis. People might race for it and that is intended.
> **Accidentally racy**: You turn off the traffic lights at a crossing. Suddenly, access to the shared resource (the crossing) is not managed anymore such that everyone rushes into it, blocking the entire box, making it impossible for anyone to move, leading to a complete traffic break-down.

If the system and model is intentionally racy, there is nothing you can do about the rejections. They must be there as they are the result of resolving the race for resources. But often, you will find the situation that the model is accidentally racy, which can be fixed by changing the model slightly. In many cases, contention arises due to contract-keys being fetched and updated. The issue is then that the transaction is built in phase 1, looking at the contract key state at that time. The validation / conflict detection happens then in phase 3. If any other transaction changed that particular key in the time between phase 1 and phase 3, the transaction will fail.

Whether you get `INACTIVE_CONTRACTS`, `LOCKED_CONTRACT` or `CONTRACT_NOT_FOUND` just depends on timing of the competing transaction. `LOCKED` means: there is a transaction about to change this resource, but we have not yet received the final verdict on it.

1.34.10.2 How To Change Your Model To Avoid Undesired Contention

Now, you can resolve such accidental raciness by introducing order into the race. As an example, you let individuals submit request contracts and you add one delegated party that receives these requests and orders and performs their application to a shared resource (through delegation). As an example, if you have an AccountIdGenerator:

```
template AccountIdGenerator
        next : Integer
        …
where
        choice NextAccount : (ContractId AccountId, ContractId AccountIdGenerator)
```

(continues on next page)

(continued from previous page)

```
        do
                a <- create this with next = next + 1
                b <- create AccountId with accountId = next
                return (a,b)
```

This `AccountIdGenerator` contract will be very racy. However, you can just add a:

```
template GetAccountIdRequest
```

and then have a single application consume these requests and generate ids. That single application knows whether it has already spent the existing `AccountIdGenerator` contract. Of course, it would make sense to support a list of requests in the choice `NextAccount` such that many AccountIds are created at once, as otherwise, the throughput of account allocation would be limited.

This is just a simple example, but should be sufficient to illustrate the issue and the solution idea.

1.34.10.3 How To Find Contention

In a distributed application, where different systems such as Triggers, Nanobots, Ingestion Application etc submit transactions, it is often not easy to understand where the contention is coming from. Here is a recipe that can be used on the Canton level:

1. Ensure that you have turned on Detailed API Logging with Debug logs.
2. Run your system / tests until you have collected enough information / rejections.
3. Open the log files and search for one of the rejections, i.e. search for `LOCKED`.
4. Filter by the trace-id of this rejection. Determine the command-id using the rosetta stone log entry. Add the command-id to the filter.
5. Now, find the ApiRequestLogger log entry of the CommandSubmissionService. This log entry contains the entire command that the application has submitted (if you turned on the detailed api logging). I.e. the exercise choice that caused the contention.
6. Then, go back to the rejection (i.e. the one with `LOCKED`). This rejection will contain a `ResourceInfo`, referring to the key / contract that caused the rejection. The ResourceInfo will contain the key that caused the failure.

Using the above recipe, you determine the choice and which key in that particular choice created the problem. This should be sufficient to find the problematic parts in the model.

1.34.11 Use Bisection to Narrow Down the Root Cause

In this section an alternative approach is outlined that could help you if the guidelines in the previous sections were insufficient to resolve the problem. To apply that approach, you do not need a deep understanding of Canton. It is not only suitable to investigate problems inside of Canton, it also helps to discover problems coming from the environment.

The approach is best explained with an example. Suppose you have developed a Canton deployment and successfully tested it on your local machine. After moving it to the distributed test environment, it is showing some problems. So you have two Canton deployments, a local one and distributed one, one of them works correctly, the other one is broken.

You notice the following differences between the two deployments:

The local deployment runs all nodes in a single process. The distributed deployment runs nodes in different processes.
The local deployment runs all nodes on the same machine. The distributed deployment runs nodes on different machines.
Only the distributed deployment has TLS enabled.
Only the distributed deployment has high-availability enabled.
The distributed deployment runs in a docker container (e.g. by using a cloud environment). The local deployment does not use docker.

To better understand which of the differences is causing the problem, you setup a new deployment that has **only half of the differences**. That could mean, you setup a new deployment with the following characteristics:

It runs nodes in **different processes** (like the distributed deployment)
It runs nodes on the same machine (like the test deployment).
It has TLS **enabled**.
It has high availability disabled.
It does not use docker.

For the sake of reference, let's call it Deployment 3 . Now you rerun the test. If the test succeeds (as for the local deployment), you know that the problem in the distributed deployment is caused by the network, by high-availability, or by docker. If the test fails (as for the distributed deployment), you know that the problem is caused by running several processes, by using TLS or by both. For the sake of the illustration, let's assume the test succeeds.

To further narrow down the root cause, you setup yet another deployment that is in the middle between Deployment 3 (which was successful) and the distributed deployment (which was failing). That could mean:

It runs nodes in different processes.
It runs nodes on **different** machines (like the distributed deployment).
It has TLS enabled.
It has high availability disabled.
It does not use docker.

Let's call it Deployment 4 . Again, you rerun the test. If the test succeeds, you know that the problem in the distributed deployment is caused by high-availability or by docker. If the test fails, you know that the problem is caused by some combination of running nodes in different processes, on different machines and having TLS enabled. Let's assume that the test fails.

To further narrow down the root cause, try to set up the simplest possible deployment that still has the problem. That could mean:

You simplify your test, e.g., **run a ping** instead of a complex workflow. It runs **only two nodes** (because you are aiming for a minimal example).
The two nodes run **in different processes on different machines** (because that seemed to be the root cause).
TLS is **disabled** (because that seemed not to trigger the problem).
High availability is disabled.
It does not use docker.

Let's call it Deployment 5 . If the test fails on Deployment 5 , you have a minimal example to reproduce the problem. You know that the problem is caused by running two nodes on different machines. The problem is independent of your DAML workflow, occurs already with two nodes and without enabling TLS. If the test succeeds on Deployment 5 , you have not yet understood the root

cause. In that case, you need to do yet another iteration with a deployment in the middle between Deployment 4 and Deployment 5 .

The following guidelines are helpful to make this approach successful:

Try to keep the list of differences between successful and failing deployment **as complete as possible**. If the root cause is not on your list, you can't find it. Differences can come from configuration, DAML models, ledger applications, deployment (in process, network, docker, kubernetes,), hardware, operating system.
Always **aim at the middle** between the successful and failing deployment to learn the most with every new deployment you create and test. That is the fastest path to the root cause.
Don't make assumptions up front of which difference may or may not cause the problem. For example, if you are making the assumption that the problem is not caused by TLS, you may save one iteration, if you are right. But you will take a long detour, if you are wrong.
Do not assume that the problem is caused by a single difference between the two deployments. It could very well be that a **combination of differences** is needed to **reproduce the problem**.

1.35 Error Codes

1.35.1 Overview

The goal of all error messages in Daml is to enable users, developers, and operators to act independently on the encountered errors, either manually or with an automated process.

Most errors are a result of request processing. Each error is logged and returned to the user as a failed gRPC response containing the status code, an optional status message and optional metadata. We further enhance this by providing:

improved consistency of the returned errors across API endpoints
richer error payload format with clearly distinguished machine readable parts to facilitate automated error handling strategies
complete inventory of all error codes with an explanation, suggested resolution and other useful information

1.35.2 Glossary

Error Represents an occurrence of a failure. Consists of:
an *error code id*,
a gRPC status code (determined by its error category),
an *error category*,
a *correlation id*,
a human readable message,
and optional additional metadata.
You can think of it as an instantiation of an error code.

Error Code Represents a class of failures. Identified by its error code id (we may use *error code* and *error code id* interchangeably in this document). Belongs to a single error category.

Error Category A broad categorization of error codes that you can base your error handling strategies on. Maps to exactly one gRPC status code. We recommended dealing with errors based on their error category. However, if the error category alone is too generic you can act on a particular error codes.

Correlation Id A value that allows the user to clearly identify the request, such that the operator can lookup any log information associated with this error. We use the request's submission id for correlation id.

1.35.3 Anatomy of an Error

Errors returned to users contain a gRPC status code, a description, and additional machine-readable information represented in the rich gRPC error model.

1.35.3.1 Error Description

We use the standard gRPC description that additionally adheres to our custom message format:

```
<ERROR_CODE_ID>(<CATEGORY_ID>,<CORRELATION_ID_PREFIX>):<HUMAN_READABLE_MESSAGE>
```

The constituent parts are:

<ERROR_CODE_ID> - a unique non-empty string containing at most 63 characters: upper-case letters, underscores or digits. Identifies corresponding error code id.
<CATEGORY_ID> - a small integer identifying the corresponding error category.
<CORRELATION_ID_PREFIX> - a string aimed at identifying originating request. Absence of one is indicated by value 0. If present, it is the first 8 characters of the corresponding request's submission id. Full correlation id can be found in error's additional machine readable information (see *Additional Machine-Readable Information*).
: - a colon serves as a separator for the machine and human readable parts of the error description.
<HUMAN_READABLE_MESSAGE> - a message targeted at a human reader. Should never be parsed by applications, as the description might change in future releases to improve clarity.

In a concrete example an error description might look like this:

```
TRANSACTION_NOT_FOUND(11,12345): Transaction not found, or not visible.
```

1.35.3.2 Additional Machine-Readable Information

We use following error details:

A mandatory com.google.rpc.ErrorInfo containing *error code id*.
A mandatory com.google.rpc.RequestInfo containing (not-truncated) correlation id (or 0 if correlation id is not available).
An optional com.google.rpc.RetryInfo containing retry interval with milliseconds resolution.
An optional com.google.rpc.ResourceInfo containing information about the resource the failure is based on. Any request that fails due to some well-defined resource issues (such as contract, contract-key, package, party, template, domain, etc.) contains these. Particular resources are implementation specific and vary across ledger implementations.

Many errors will include more information, but there is no guarantee that additional information will be preserved across versions.

1.35.3.3 Prevent Security Leaks in Error Codes

For any error that could leak information to an attacker, the system returns an error message via the API that contains no valuable information. The log file contains the full error message.

1.35.3.4 Work With Error Codes

This example shows how a user can extract the relevant error information:

```
object SampleClientSide {

  import com.google.rpc.ResourceInfo
  import com.google.rpc.{ErrorInfo, RequestInfo, RetryInfo}
  import io.grpc.StatusRuntimeException
  import scala.jdk.CollectionConverters._

  def example(): Unit = {
    try {
      DummmyServer.serviceEndpointDummy()
    } catch {
      case e: StatusRuntimeException =>
        // Converting to a status object.
        val status = io.grpc.protobuf.StatusProto.fromThrowable(e)

        // Extracting gRPC status code.
        assert(status.getCode == io.grpc.Status.Code.ABORTED.value())
        assert(status.getCode == 10)

        // Extracting error message, both
        // machine oriented part: "MY_ERROR_CODE_ID(2,full-cor):",
        // and human oriented part: "A user oriented message".
        assert(status.getMessage == "MY_ERROR_CODE_ID(2,full-cor): A user
↪oriented message")

        // Getting all the details
        val rawDetails: Seq[com.google.protobuf.Any] = status.getDetailsList.
↪asScala.toSeq

        // Extracting error code id, error category id and optionally additional
↪metadata.
        assert {
          rawDetails.collectFirst {
            case any if any.is(classOf[ErrorInfo]) =>
              val v = any.unpack(classOf[ErrorInfo])
              assert(v.getReason == "MY_ERROR_CODE_ID")
              assert(v.getMetadataMap.asScala.toMap == Map("category" -> "2", "foo
↪" -> "bar"))
          }.isDefined
        }

        // Extracting full correlation id, if present.
        assert {
          rawDetails.collectFirst {
            case any if any.is(classOf[RequestInfo]) =>
              val v = any.unpack(classOf[RequestInfo])
```

(continues on next page)

(continued from previous page)

```
              assert(v.getRequestId == "full-correlation-id-123456790")
          }.isDefined
        }

        // Extracting retry information if the error is retryable.
        assert {
          rawDetails.collectFirst {
            case any if any.is(classOf[RetryInfo]) =>
              val v = any.unpack(classOf[RetryInfo])
              assert(v.getRetryDelay.getSeconds == 123, v.getRetryDelay.
↪getSeconds)
              assert(v.getRetryDelay.getNanos == 456 * 1000 * 1000, v.
↪getRetryDelay.getNanos)
          }.isDefined
        }

        // Extracting resource if the error pertains to some well defined
↪resource.
        assert {
          rawDetails.collectFirst {
            case any if any.is(classOf[ResourceInfo]) =>
              val v = any.unpack(classOf[ResourceInfo])
              assert(v.getResourceType == "CONTRACT_ID")
              assert(v.getResourceName == "someContractId")
          }.isDefined
        }
    }
  }
}
```

1.35.4 Error Codes In Canton Operations

Almost all errors and warnings that can be generated by a Canton-based system are annotated with error codes of the form **SOMETHING_NOT_SO_GOOD_HAPPENED(x,c)**. The upper case string with underscores denotes the unique error id. The parentheses include key additional information. The id together with the extra information is referred to as `error-code`. The **x** represents the *ErrorCategory* used to classify the error, and the **c** represents the first 8 characters of the correlation id associated with this request, or 0 if no correlation id is given.

The majority of errors in Canton-based systems are a result of request processing and are logged and returned to the user as described above. In other cases, errors occur due to background processes (i.e. network connection issues/transaction confirmation processing). Such errors are only logged.

Generally, we use the following log levels on the server:

INFO to log user errors where the error leads to a failure of the request but the system remains healthy.
WARN to log degradations of the system or point out unusual behaviour.
ERROR to log internal errors where the system does not behave properly and immediate attention is required.

On the client side, failures are considered to be errors and logged as such.

1.35.5 Error Categories

The error categories allow you to group errors such that application logic can be built to automatically deal with errors and decide whether to retry a request or escalate to the operator.

A full list of error categories is documented here.

1.35.6 Machine Readable Information

Every error on the API is constructed to allow automated and manual error handling. First, the error category maps to exactly one gRPC status code. Second, every error description (of the corresponding `StatusRuntimeException.Status`) starts with the error information (**SOMETHING_NOT_SO_GOOD_HAPPENED(CN,x)**), separated from a human readable description by a colon (:). The rest of the description is targeted to humans and should never be parsed by applications, as the description might change in future releases to improve clarity.

In addition to the status code and the description, the gRPC rich error model is used to convey additional, machine-readable information to the application.

Therefore, to support automatic error processing, an application may:

parse the error information from the beginning of the description to obtain the error-id, the error category and the component.
use the gRPC-code to get the set of possible error categories.
if present, use the `ResourceInfo` included as `Status.details`. Any request that fails due to some well-defined resource issues (contract, contract-key, package, party, template, domain) will contain these, calling out on what resource the failure is based on.
use the `RetryInfo` to determine the recommended retry interval (or make this decision based on the category / gRPC code).
use the `RequestInfo.id` as the *correlation-id*, included as `Status.details`.
use the `ErrorInfo.reason` as error-id and `ErrorInfo.metadata("category")` as error category, included as `Status.details`.

All this information is included in errors that are generated by components under our control and included as `Status.details`. As with Daml error codes described above, many errors include more information, but there is no guarantee that additional information will be preserved across versions.

Generally, automated error handling can be done on any level (e.g. load balancer using gRPC status codes, application using `ErrorCategory` or human reacting to error-ids). In most cases it is advisable to deal with errors on a per category basis and deal with error-ids in very specific situations which are application dependent. For example, a command failure with the message CONTRACT_NOT_FOUND may be an application failure in case the given application is the only actor on the contracts, whereas a CONTRACT_NOT_FOUND message is to be expected in a case where multiple independent actors operate on the ledger state.

1.35.7 Example

If an application submits a Daml transaction that exceeds the size limits enforced on a domain, the command will be rejected. Using the logs of one of our test cases, the participant node will log the following message:

```
2022-04-26 11:37:54,584 [GracefulRejectsIntegrationTestDefault-env-execution-
↪context-30] INFO  c.d.c.p.p.TransactionProcessingSteps:participant=participant1/
↪domain=da tid:13617c1bda402e54e016a6a17637cb20 - SEQUENCER_REQUEST_FAILED(2,
↪13617c1b): Failed to send command err-context:
↪{location=TransactionProcessingSteps.scala:449, sendError=RequestInvalid(Batch
↪size (85134 bytes) is exceeding maximum size (27000 bytes) for domain
↪da::12201253c344...)}
```

The machine-readable part of the error message appears as `SEQUENCER_REQUEST_FAILED(2, 13617c1b)`, mentioning the error id `SEQUENCER_REQUEST_FAILED`, the category *ContentionOnSharedResources* with `id=2`, and the correlation identifier `13617c1b`. Please note that there is no guarantee on the name of the logger that is emitting the given error, as this name is internal and subject to change. The human-readable part of the log message should not be parsed, as we might subsequently improve the text.

The client will receive the error information as a Grpc error:

```
2022-04-26 11:37:54,923 [ScalaTest-run-running-
↪GracefulRejectsIntegrationTestDefault] ERROR  c.d.c.i.
↪EnterpriseEnvironmentDefinition$$anon$3 - Request failed for participant1.
    GrpcRequestRefusedByServer: ABORTED/SEQUENCER_REQUEST_FAILED(2,13617c1b):
↪Failed to send command
    Request: SubmitAndWaitTransactionTree(actAs = participant1::1220baa5cd30...,
↪commandId = '', workflowId = '', submissionId = '', deduplicationPeriod =
↪None(), ledgerId = 'participant1', commands= ...)
    CorrelationId: 13617c1bda402e54e016a6a17637cb20
    RetryIn: 1 second
    Context: HashMap(participant -> participant1, test ->
↪GracefulRejectsIntegrationTestDefault, domain -> da, sendError ->
↪RequestInvalid(Batch size (85134 bytes) is exceeding maximum size (27000 bytes)
↪for domain da::12201253c344...), definite_answer -> true)
```

Note that the second log is created by Daml tooling that prints the Grpc Status into the log files during tests. The actual Grpc error would be received by the application and would not be logged by the participant node in the given form.

1.35.8 Error Categories Inventory

The error categories allow you to group errors such that application logic can be built in a sensible way to automatically deal with errors and decide whether to retry a request or escalate to the operator.

1.35.8.1 TransientServerFailure

Category id: 1

gRPC status code: UNAVAILABLE

Default log level: INFO

Description: One of the services required to process the request was not available.

Resolution: Expectation: transient failure that should be handled by retrying the request with appropriate backoff.

Retry strategy: Retry quickly in load balancer.

1.35.8.2 ContentionOnSharedResources

Category id: 2

gRPC status code: ABORTED

Default log level: INFO

Description: The request could not be processed due to shared processing resources (e.g. locks or rate limits that replenish quickly) being occupied. If the resource is known (i.e. locked contract), it will be included as a resource info. (Not known resource contentions are e.g. overloaded networks where we just observe timeouts, but can't pin-point the cause).

Resolution: Expectation: this is processing-flow level contention that should be handled by retrying the request with appropriate backoff.

Retry strategy: Retry quickly (indefinitely or limited), but do not retry in load balancer.

1.35.8.3 DeadlineExceededRequestStateUnknown

Category id: 3

gRPC status code: DEADLINE_EXCEEDED

Default log level: INFO

Description: The request might not have been processed, as its deadline expired before its completion was signalled. Note that for requests that change the state of the system, this error may be returned even if the request has completed successfully. Note that known and well-defined timeouts are signalled as [[ContentionOnSharedResources]], while this category indicates that the state of the request is unknown.

Resolution: Expectation: the deadline might have been exceeded due to transient resource congestion or due to a timeout in the request processing pipeline being too low.

The transient errors might be solved by the application retrying. The non-transient errors will require operator intervention to change the timeouts.

Retry strategy: Retry for a limited number of times with deduplication.

1.35.8.4 SystemInternalAssumptionViolated

Category id: 4

gRPC status code: INTERNAL

Default log level: ERROR

Description: Request processing failed due to a violation of system internal invariants. This error is exposed on the API with grpc-status INTERNAL without any details for security reasons

Resolution: Expectation: this is due to a bug in the implementation or data corruption in the systems databases. Resolution will require operator intervention, and potentially vendor support.

Retry strategy: Retry after operator intervention.

1.35.8.5 AuthInterceptorInvalidAuthenticationCredentials

Category id: 6

gRPC status code: UNAUTHENTICATED

Default log level: WARN

Description: The request does not have valid authentication credentials for the operation. This error is exposed on the API with grpc-status UNAUTHENTICATED without any details for security reasons

Resolution: Expectation: this is an application bug, application misconfiguration or ledger-level misconfiguration. Resolution requires application and/or ledger operator intervention.

Retry strategy: Retry after application operator intervention.

1.35.8.6 InsufficientPermission

Category id: 7

gRPC status code: PERMISSION_DENIED

Default log level: WARN

Description: The caller does not have permission to execute the specified operation. This error is exposed on the API with grpc-status PERMISSION_DENIED without any details for security reasons

Resolution: Expectation: this is an application bug or application misconfiguration. Resolution requires application operator intervention.

Retry strategy: Retry after application operator intervention.

1.35.8.7 SecurityAlert

Category id: 5

gRPC status code: INVALID_ARGUMENT

Default log level: WARN

Description: A potential attack or a faulty peer component has been detected. This error is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.

Resolution: Expectation: this can be a severe issue that requires operator attention or intervention, and potentially vendor support. It means that the system has detected invalid information that can be attributed to either faulty or malicious manipulation of data coming from a peer source.

Retry strategy: Errors in this category are non-retryable.

1.35.8.8 InvalidIndependentOfSystemState

Category id: 8

gRPC status code: INVALID_ARGUMENT

Default log level: INFO

Description: The request is invalid independent of the state of the system.

Resolution: Expectation: this is an application bug or ledger-level misconfiguration (e.g. request size limits). Resolution requires application and/or ledger operator intervention.

Retry strategy: Retry after application operator intervention.

1.35.8.9 InvalidGivenCurrentSystemStateOther

Category id: 9

gRPC status code: FAILED_PRECONDITION

Default log level: INFO

Description: The mutable state of the system does not satisfy the preconditions required to execute the request. We consider the whole Daml ledger including ledger config, parties, packages, users and command deduplication to be mutable system state. Thus all Daml interpretation errors are reported as this error or one of its specializations.

Resolution: ALREADY_EXISTS and NOT_FOUND are special cases for the existence and non-existence of well-defined entities within the system state; e.g., a .dalf package, contracts ids, contract keys, or a transaction at an offset. OUT_OF_RANGE is a special case for reading past a range. Violations of the Daml ledger model always result in these kinds of errors. Expectation: this is due to application-level bugs, misconfiguration or contention on application-visible resources; and might be resolved by retrying later, or after changing the state of the system. Handling these errors requires an application-specific strategy and/or operator intervention.

Retry strategy: Retry after application operator intervention.

1.35.8.10 InvalidGivenCurrentSystemStateResourceExists

Category id: 10

gRPC status code: ALREADY_EXISTS

Default log level: INFO

Description: Special type of InvalidGivenCurrentSystemState referring to a well-defined resource.

Resolution: Same as [[InvalidGivenCurrentSystemStateOther]].

Retry strategy: Inspect resource failure and retry after resource failure has been resolved (depends on type of resource and application).

1.35.8.11 InvalidGivenCurrentSystemStateResourceMissing

Category id: 11

gRPC status code: NOT_FOUND

Default log level: INFO

Description: Special type of InvalidGivenCurrentSystemState referring to a well-defined resource.

Resolution: Same as [[InvalidGivenCurrentSystemStateOther]].

Retry strategy: Inspect resource failure and retry after resource failure has been resolved (depends on type of resource and application).

1.35.8.12 InvalidGivenCurrentSystemStateSeekAfterEnd

Category id: 12

gRPC status code: OUT_OF_RANGE

Default log level: INFO

Description: This error is only used by the Ledger API server in connection with invalid offsets.

Resolution: Expectation: this error is only used by the Ledger API server in connection with invalid offsets.

Retry strategy: Retry after application operator intervention.

1.35.8.13 BackgroundProcessDegradationWarning

Category id: 13

gRPC status code: N/A

Default log level: WARN

Description: This error category is used internally to signal to the system operator an internal degradation.

Resolution: Inspect details of the specific error for more information.

Retry strategy: Not an API error, therefore not retryable.

1.35.8.14 InternalUnsupportedOperation

Category id: 14

gRPC status code: UNIMPLEMENTED

Default log level: ERROR

Description: This error category is used to signal that an unimplemented code-path has been triggered by a client or participant operator request. This error is exposed on the API with grpc-status UNIMPLEMENTED without any details for security reasons

Resolution: This error is caused by a ledger-level misconfiguration or by an implementation bug. Resolution requires participant operator intervention.

Retry strategy: Errors in this category are non-retryable.

1.35.9 Error Codes Inventory - Daml

1.35.9.1 1. ParticipantErrorGroup

1.35.9.2 1.1. ParticipantErrorGroup / CommonErrors

Common errors raised in Daml services and components.

REQUEST_TIME_OUT

Explanation: This rejection is given when a request processing status is not known and a time-out is reached.

Category: DeadlineExceededRequestStateUnknown

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status DEADLINE_EXCEEDED including a detailed error message.

Resolution: Retry for transient problems. If non-transient contact the operator as the time-out limit might be too short.

SERVER_IS_SHUTTING_DOWN

Explanation: This rejection is given when the participant server is shutting down.

Category: TransientServerFailure

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status UNAVAILABLE including a detailed error message.

Resolution: Contact the participant operator.

SERVICE_INTERNAL_ERROR

Explanation: This error occurs if one of the services encountered an unexpected exception.

Category: SystemInternalAssumptionViolated

Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.

Resolution: Contact support.

SERVICE_NOT_RUNNING

Explanation: This rejection is given when the requested service has already been closed.

Category: TransientServerFailure

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status UNAVAILABLE including a detailed error message.

Resolution: Retry re-submitting the request. If the error persists, contact the participant operator.

UNSUPPORTED_OPERATION

Explanation: This error category is used to signal that an unimplemented code-path has been triggered by a client or participant operator request.

Category: InternalUnsupportedOperation

Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status UNIMPLEMENTED without any details for security reasons.

Resolution: This error is caused by a participant node misconfiguration or by an implementation bug. Resolution requires participant operator intervention.

1.35.9.3 1.2. ParticipantErrorGroup / IndexErrors

Errors raised by the Participant Index persistence layer.

1.35.9.4 1.2.1. ParticipantErrorGroup / IndexErrors / DatabaseErrors

INDEX_DB_INVALID_RESULT_SET

Explanation: This error occurs if the result set returned by a query against the Index database is invalid.

Category: SystemInternalAssumptionViolated

Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.

Resolution: Contact support.

INDEX_DB_SQL_NON_TRANSIENT_ERROR

Explanation: This error occurs if a non-transient error arises when executing a query against the index database.

Category: SystemInternalAssumptionViolated

Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.

Resolution: Contact the participant operator.

INDEX_DB_SQL_TRANSIENT_ERROR

Explanation: This error occurs if a transient error arises when executing a query against the index database.

Category: TransientServerFailure

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status UNAVAILABLE including a detailed error message.

Resolution: Re-submit the request.

1.35.9.5 1.3. ParticipantErrorGroup / LedgerApiErrors

Errors raised by or forwarded by the Ledger API.

HEAP_MEMORY_OVER_LIMIT

Explanation: This error happens when the JVM heap memory pool exceeds a pre-configured limit.

Category: ContentionOnSharedResources

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.

Resolution: The following actions can be taken: 1. Review the historical use of heap space by inspecting the metric given in the message. 2. Review the current heap space limits configured in the rate limiting configuration. 3. Try to space out requests that are likely to require a large amount of memory to process.

LEDGER_API_INTERNAL_ERROR

Explanation: This error occurs if there was an unexpected error in the Ledger API.

Category: SystemInternalAssumptionViolated

Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.

Resolution: Contact support.

MAXIMUM_NUMBER_OF_STREAMS

Explanation: This error happens when the number of concurrent gRPC streaming requests exceeds the configured limit.

Category: ContentionOnSharedResources

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.

Resolution: The following actions can be taken: 1. Review the historical need for concurrent streaming by inspecting the metric given in the message. 2. Review the maximum streams limit configured in the rate limiting configuration. 3. Try to space out streaming requests such that they do not need to run in parallel with each other.

PARTICIPANT_BACKPRESSURE

Explanation: This error occurs when a participant rejects a command due to excessive load. Load can be caused by the following factors: 1. when commands are submitted to the participant through its Ledger API, 2. when the participant receives requests from other participants through a connected domain.

Category: ContentionOnSharedResources

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.

Resolution: Wait a bit and retry, preferably with some backoff factor. If possible, ask other participants to send fewer requests; the domain operator can enforce this by imposing a rate limit.

THREADPOOL_OVERLOADED

Explanation: This happens when the rate of submitted gRPC requests requires more CPU or database power than is available.

Category: ContentionOnSharedResources

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.

Resolution: The following actions can be taken: Here the 'queue size' for the threadpool is considered as reported by the executor itself. 1. Review the historical 'queue size' growth by inspecting the metric given in the message. 2. Review the maximum 'queue size' limits configured in the rate limiting configuration. 3. Try to space out requests that are likely to require a lot of CPU or database power.

1.35.9.6 1.3.1. ParticipantErrorGroup / LedgerApiErrors / AdminServices

Errors raised by Ledger API admin services.

CONFIGURATION_ENTRY_REJECTED

Explanation: This rejection is given when a new configuration is rejected.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: Fetch newest configuration and/or retry.

INTERNALLY_INVALID_KEY

Explanation: A cryptographic key used by the configured system is not valid

Category: SystemInternalAssumptionViolated

Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.

Resolution: Contact support.

PACKAGE_UPLOAD_REJECTED

Explanation: This rejection is given when a package upload is rejected.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: Refer to the detailed message of the received error.

1.35.9.7 1.3.1.1. ParticipantErrorGroup / LedgerApiErrors / AdminServices / IdentityProviderConfigServiceErrorGroup

IDP_CONFIG_ALREADY_EXISTS

Explanation: There already exists an identity provider configuration with the same identity provider id.

Category: InvalidGivenCurrentSystemStateResourceExists

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ALREADY_EXISTS including a detailed error message.

Resolution: Check that you are connecting to the right participant node and the identity provider id is spelled correctly, or use an identity provider that already exists.

IDP_CONFIG_BY_ISSUER_NOT_FOUND

Explanation: The identity provider config referred to by the request was not found.

Category: InvalidGivenCurrentSystemStateResourceMissing

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.

Resolution: Check that you are connecting to the right participant node and the identity provider config is spelled correctly, or create the configuration.

IDP_CONFIG_ISSUER_ALREADY_EXISTS

Explanation: There already exists an identity provider configuration with the same issuer.

Category: InvalidGivenCurrentSystemStateResourceExists

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ALREADY_EXISTS including a detailed error message.

Resolution: Check that you are connecting to the right participant node and the identity provider id is spelled correctly, or use an identity provider that already exists.

IDP_CONFIG_NOT_FOUND

Explanation: The identity provider config referred to by the request was not found.

Category: InvalidGivenCurrentSystemStateResourceMissing

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.

Resolution: Check that you are connecting to the right participant node and the identity provider config is spelled correctly, or create the configuration.

INVALID_IDENTITY_PROVIDER_UPDATE_REQUEST

Explanation: There was an attempt to update an identity provider config using an invalid update request.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Inspect the error details for specific information on what made the request invalid. Retry with an adjusted update request.

TOO_MANY_IDENTITY_PROVIDER_CONFIGS

Explanation: A system can have only a limited number of identity provider configurations. There was an attempt to create an identity provider configuration.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: Delete some of the already existing identity provider configurations. Contact the participant operator if the limit is too low.

1.35.9.8 1.3.1.2. ParticipantErrorGroup / LedgerApiErrors / AdminServices / PartyManagementServiceErrorGroup

CONCURRENT_PARTY_DETAILS_UPDATE_DETECTED

Explanation: Concurrent updates to a party can be controlled by supplying an update request with a resource version (this is optional). A party's resource version can be obtained by reading the party on the Ledger API. There was attempt to update a party using a stale resource version, indicating that a different process had updated the party earlier.

Category: ContentionOnSharedResources

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.

Resolution: Read this party again to obtain its most recent state and in particular its most recent resource version. Use the obtained information to build and send a new update request.

INTERNAL_PARTY_RECORD_ALREADY_EXISTS

Explanation: Each on-ledger party known to this participant node can have a participant's local metadata assigned to it. The local information about a party referred to by this request was found when it should have been not found.

Category: SystemInternalAssumptionViolated

Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.

Resolution: This error can indicate a problem with the server's storage or implementation.

INTERNAL_PARTY_RECORD_NOT_FOUND

Explanation: Each on-ledger party known to this participant node can have a participant's local metadata assigned to it. The local information about a party referred to by this request was not found when it should have been found.

Category: SystemInternalAssumptionViolated

Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.

Resolution: This error can indicate a problem with the server's storage or implementation.

INVALID_PARTY_DETAILS_UPDATE_REQUEST

Explanation: There was an attempt to update a party using an invalid update request.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Inspect the error details for specific information on what made the request invalid. Retry with an adjusted update request.

MAX_PARTY_DETAILS_ANNOTATIONS_SIZE_EXCEEDED

Explanation: A party can have at most 256kb worth of annotations in total measured in number of bytes in UTF-8 encoding. There was an attempt to allocate or update a party such that this limit would have been exceeded.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: Retry with fewer annotations or delete some of the party's existing annotations.

PARTY_NOT_FOUND

Explanation: The party referred to by the request was not found.

Category: InvalidGivenCurrentSystemStateResourceMissing

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.

Resolution: Check that you are connecting to the right participant node and that the party is spelled correctly.

1.35.9.9 1.3.1.3. ParticipantErrorGroup / LedgerApiErrors / AdminServices / UserManagementServiceErrorGroup

CONCURRENT_USER_UPDATE_DETECTED

Explanation: Concurrent updates to a user can be controlled by supplying an update request with a resource version (this is optional). A user's resource version can be obtained by reading the user on the Ledger API. There was attempt to update a user using a stale resource version, indicating that a different process had updated the user earlier.

Category: ContentionOnSharedResources

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.

Resolution: Read this user again to obtain its most recent state and in particular its most recent resource version. Use the obtained information to build and send a new update request.

INVALID_USER_UPDATE_REQUEST

Explanation: There was an attempt to update a user using an invalid update request.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Inspect the error details for specific information on what made the request invalid. Retry with an adjusted update request.

MAX_USER_ANNOTATIONS_SIZE_EXCEEDED

Explanation: A user can have at most 256kb worth of annotations in total measured in number of bytes in UTF-8 encoding. There was an attempt to create or update a user such that this limit would have been exceeded.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: Retry with fewer annotations or delete some of the user's existing annotations.

TOO_MANY_USER_RIGHTS

Explanation: A user can have only a limited number of user rights. There was an attempt to create a user with too many rights or grant too many rights to a user.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: Retry with a smaller number of rights or delete some of the already existing rights of this user. Contact the participant operator if the limit is too low.

USER_ALREADY_EXISTS

Explanation: There already exists a user with the same user-id.

Category: InvalidGivenCurrentSystemStateResourceExists

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ALREADY_EXISTS including a detailed error message.

Resolution: Check that you are connecting to the right participant node and the user-id is spelled correctly, or use the user that already exists.

USER_NOT_FOUND

Explanation: The user referred to by the request was not found.

Category: InvalidGivenCurrentSystemStateResourceMissing

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.

Resolution: Check that you are connecting to the right participant node and the user-id is spelled correctly, if yes, create the user.

1.35.9.10 1.3.2. ParticipantErrorGroup / LedgerApiErrors / AuthorizationChecks

Authentication and authorization errors.

INTERNAL_AUTHORIZATION_ERROR

Explanation: An internal system authorization error occurred.

Category: SystemInternalAssumptionViolated

Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.

Resolution: Contact the participant operator.

PERMISSION_DENIED

Explanation: This rejection is given if the supplied authorization token is not sufficient for the intended command. The exact reason is logged on the participant, but not given to the user for security reasons.

Category: InsufficientPermission

Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status PERMISSION_DENIED without any details for security reasons.

Resolution: Inspect your command and your token or ask your participant operator for an explanation why this command failed.

STALE_STREAM_AUTHORIZATION

Explanation: The stream was aborted because the authenticated user's rights changed, and the user might thus no longer be authorized to this stream.

Category: ContentionOnSharedResources

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.

Resolution: The application should automatically retry fetching the stream. It will either succeed, or fail with an explicit denial of authentication or permission.

UNAUTHENTICATED

Explanation: This rejection is given if the submitted command does not contain a JWT token on a participant enforcing JWT authentication.

Category: AuthInterceptorInvalidAuthenticationCredentials

Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status UNAUTHENTICATED without any details for security reasons.

Resolution: Ask your participant operator to provide you with an appropriate JWT token.

1.35.9.11 1.3.3. ParticipantErrorGroup / LedgerApiErrors / CommandExecution

Errors raised during the command execution phase of the command submission evaluation.

FAILED_TO_DETERMINE_LEDGER_TIME

Explanation: This error occurs if the participant fails to determine the max ledger time of the used contracts. Most likely, this means that one of the contracts is not active anymore which can happen under contention. It can also happen with contract keys.

Category: ContentionOnSharedResources

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.

Resolution: Retry the transaction submission.

1.35.9.12 1.3.3.1. ParticipantErrorGroup / LedgerApiErrors / CommandExecution / Interpreter

Errors raised during the command interpretation phase of the command submission evaluation.

CONTRACT_DOES_NOT_IMPLEMENT_INTERFACE

Explanation: This error occurs when you try to coerce/use a contract via an interface that it does not implement.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Ensure the contract you are calling does implement the interface you are using to do so. Avoid writing LF/low-level interface implementation classes manually.

CONTRACT_DOES_NOT_IMPLEMENT_REQUIRING_INTERFACE

Explanation: This error occurs when you try to create/use a contract that does not implement the requiring interfaces of some other interface that it does implement.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Ensure you implement all required interfaces correctly, and avoid writing LF/low-level interface implementation classes manually.

CONTRACT_ID_COMPARABILITY

Explanation: This error occurs when you attempt to compare a global and local contract ID of the same discriminator.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Avoid constructing contract IDs manually.

CONTRACT_ID_IN_CONTRACT_KEY

Explanation: This error occurs when a contract key contains a contract ID, which is illegal for hashing reasons.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Ensure your contracts key field cannot contain a contract ID.

CONTRACT_NOT_ACTIVE

Explanation: This error occurs if an exercise or fetch happens on a transaction-locally consumed contract.

Category: InvalidGivenCurrentSystemStateResourceMissing

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.

Resolution: This error indicates an application error.

CREATE_EMPTY_CONTRACT_KEY_MAINTAINERS

Explanation: This error occurs when you try to create a contract that has a key, but with empty maintainers.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Check the definition of the contract key's maintainers, and ensure this list won't be empty given your creation arguments.

DAML_AUTHORIZATION_ERROR

Explanation: This error occurs if a Daml transaction fails due to an authorization error. An authorization means that the Daml transaction computed a different set of required submitters than you have provided during the submission as *actAs* parties.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: This error type occurs if there is an application error.

DAML_INTERPRETATION_ERROR

Explanation: This error occurs if a Daml transaction fails during interpretation.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: This error type occurs if there is an application error.

DAML_INTERPRETER_INVALID_ARGUMENT

Explanation: This error occurs if a Daml transaction fails during interpretation due to an invalid argument.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: This error type occurs if there is an application error.

DISCLOSED_CONTRACT_KEY_HASHING_ERROR

Explanation: This error occurs if a user attempts to provide a key hash for a disclosed contract which we have already cached to be different.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: Ensure the contract ID and contract payload you have provided in your disclosed contract is correct.

FETCH_EMPTY_CONTRACT_KEY_MAINTAINERS

Explanation: This error occurs when you try to fetch a contract by key, but that key would have empty maintainers.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Check the definition of the contract key's maintainers, and ensure this list won't be empty given the contract key you are fetching.

INTERPRETATION_DEV_ERROR

Explanation: This error is a catch-all for errors thrown by in-development features, and should never be thrown in production.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: See the error message for details of the specific in-development feature error. If this is production, avoid using development features.

INTERPRETATION_USER_ERROR

Explanation: This error occurs when a user calls abort or error on an LF version before native exceptions were introduced.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: Either remove the call to abort, error or perhaps assert, or ensure you are exercising your contract choice as the author expects.

NON_COMPARABLE_VALUES

Explanation: This error occurs when you attempt to compare two values of different types using the built-in comparison types.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Avoid using the low level comparison build, and instead use the Eq class.

TEMPLATE_PRECONDITION_VIOLATED

Explanation: This error occurs when a contract's pre-condition (the ensure clause) is violated on contract creation.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Ensure the contract argument you are passing into your create doesn't violate the conditions of the contract.

UNHANDLED_EXCEPTION

Explanation: This error occurs when a user throws an error and does not catch it with try-catch.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: Either your error handling in a choice body is insufficient, or you are using a contract incorrectly.

WRONGLY_TYPED_CONTRACT

Explanation: This error occurs when you try to fetch/use a contract in some way with a contract ID that doesn't match the template type on the ledger.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: Ensure the contract IDs you are using are of the type we expect on the ledger. Avoid unsafely coercing contract IDs.

1.35.9.13 1.3.3.1.1. ParticipantErrorGroup / LedgerApiErrors / CommandExecution / Interpreter / LookupErrors

Errors raised in lookups during the command interpretation phase.

CONTRACT_KEY_NOT_FOUND

Explanation: This error occurs if the Daml engine interpreter cannot resolve a contract key to an active contract. This can be caused by either the contract key not being known to the participant, or not being known to the submitting parties or the contract representing an already archived key.

Category: InvalidGivenCurrentSystemStateResourceMissing

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.

Resolution: This error type occurs if there is contention on a contract.

1.35.9.14 1.3.3.2. ParticipantErrorGroup / LedgerApiErrors / CommandExecution / Package

Command execution errors raised due to invalid packages.

ALLOWED_LANGUAGE_VERSIONS

Explanation: This error indicates that the uploaded DAR is based on an unsupported language version.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Use a DAR compiled with a language version that this participant supports.

PACKAGE_VALIDATION_FAILED

Explanation: This error occurs if a package referred to by a command fails validation. This should not happen as packages are validated when being uploaded.

Category: SecurityAlert

Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.

Resolution: Contact support.

1.35.9.15 1.3.3.3. ParticipantErrorGroup / LedgerApiErrors / CommandExecution / Preprocessing

Errors raised during command conversion to the internal data representation.

COMMAND_PREPROCESSING_FAILED

Explanation: This error occurs if a command fails during interpreter pre-processing.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Inspect error details and correct your application.

1.35.9.16 1.3.4. ParticipantErrorGroup / LedgerApiErrors / ConsistencyErrors

Potential consistency errors raised due to race conditions during command submission or returned as submission rejections by the backing ledger.

CONTRACT_NOT_FOUND

Explanation: This error occurs if the Daml engine can not find a referenced contract. This can be caused by either the contract not being known to the participant, or not being known to the submitting parties or already being archived.

Category: InvalidGivenCurrentSystemStateResourceMissing

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.

Resolution: This error type occurs if there is contention on a contract.

DISCLOSED_CONTRACT_INVALID

Explanation: This error occurs if the disclosed payload or metadata of one of the contracts does not match the actual payload or metadata of the contract.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: Re-submit the command using valid disclosed contract payload and metadata.

DUPLICATE_COMMAND

Explanation: A command with the given command id has already been successfully processed.

Category: InvalidGivenCurrentSystemStateResourceExists

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ALREADY_EXISTS including a detailed error message.

Resolution: The correct resolution depends on the use case. If the error received pertains to a submission retried due to a timeout, do nothing, as the previous command has already been accepted. If the intent is to submit a new command, re-submit using a distinct command id.

DUPLICATE_CONTRACT_KEY

Explanation: This error signals that within the transaction we got to a point where two contracts with the same key were active.

Category: InvalidGivenCurrentSystemStateResourceExists

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ALREADY_EXISTS including a detailed error message.

Resolution: This error indicates an application error.

INCONSISTENT

Explanation: At least one input has been altered by a concurrent transaction submission.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: The correct resolution depends on the business flow, for example it may be possible to proceed without an archived contract as an input, or the transaction submission may be retried to load the up-to-date value of a contract key.

INCONSISTENT_CONTRACTS

Explanation: An input contract has been archived by a concurrent transaction submission.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: The correct resolution depends on the business flow, for example it may be possible to proceed without the archived contract as an input, or a different contract could be used.

INCONSISTENT_CONTRACT_KEY

Explanation: An input contract key was re-assigned to a different contract by a concurrent transaction submission.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: Retry the transaction submission.

INVALID_LEDGER_TIME

Explanation: The ledger time of the submission violated some constraint on the ledger time.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: Retry the transaction submission.

SUBMISSION_ALREADY_IN_FLIGHT

Explanation: Another command submission with the same change ID (application ID, command ID, actAs) is already being processed.

Category: ContentionOnSharedResources

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.

Resolution: Listen to the command completion stream until a completion for the in-flight command submission is published. Alternatively, resubmit the command. If the in-flight submission has finished successfully by then, this will return more detailed information about the earlier one. If the in-flight submission has failed by then, the resubmission will attempt to record the new transaction on the ledger.

1.35.9.17 1.3.5. ParticipantErrorGroup / LedgerApiErrors / PackageServiceError

Errors raised by the Package Management Service on package uploads.

DAR_NOT_SELF_CONSISTENT

Explanation: This error indicates that the uploaded Dar is broken because it is missing internal dependencies.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Contact the supplier of the Dar.

DAR_VALIDATION_ERROR

Explanation: This error indicates that the validation of the uploaded dar failed.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Inspect the error message and contact support.

PACKAGE_SERVICE_INTERNAL_ERROR

Explanation: This error indicates an internal issue within the package service.

Category: SystemInternalAssumptionViolated

Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.

Resolution: Inspect the error message and contact support.

1.35.9.18 1.3.5.1. ParticipantErrorGroup / LedgerApiErrors / PackageServiceError / Reading

Package parsing errors raised during package upload.

DAR_PARSE_ERROR

Explanation: This error indicates that the content of the Dar file could not be parsed successfully.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Inspect the error message and contact support.

INVALID_DAR

Explanation: This error indicates that the supplied dar file was invalid.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Inspect the error message for details and contact support.

INVALID_DAR_FILE_NAME

Explanation: This error indicates that the supplied dar file name did not meet the requirements to be stored in the persistence store.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Inspect error message for details and change the file name accordingly

INVALID_LEGACY_DAR

Explanation: This error indicates that the supplied zipped dar is an unsupported legacy Dar.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Please use a more recent dar version.

INVALID_ZIP_ENTRY

Explanation: This error indicates that the supplied zipped dar file was invalid.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Inspect the error message for details and contact support.

ZIP_BOMB

Explanation: This error indicates that the supplied zipped dar is regarded as zip-bomb.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Inspect the dar and contact support.

1.35.9.19 1.3.6. ParticipantErrorGroup / LedgerApiErrors / RequestValidation

Validation errors raised when evaluating requests in the Ledger API.

INVALID_ARGUMENT

Explanation: This error is emitted when a submitted ledger API command contains an invalid argument.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Inspect the reason given and correct your application.

INVALID_DEDUPLICATION_PERIOD

Explanation: This error is emitted when a submitted ledger API command specifies an invalid deduplication period.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: Inspect the error message, adjust the value of the deduplication period or ask the participant operator to increase the maximum deduplication period.

INVALID_FIELD

Explanation: This error is emitted when a submitted ledger API command contains a field value that cannot be understood.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Inspect the reason given and correct your application.

LEDGER_ID_MISMATCH

Explanation: Every ledger API command contains a ledger-id which is verified against the running ledger. This error indicates that the provided ledger-id does not match the expected one.

Category: InvalidGivenCurrentSystemStateResourceMissing

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.

Resolution: Ensure that your application is correctly configured to use the correct ledger.

MISSING_FIELD

Explanation: This error is emitted when a mandatory field is not set in a submitted ledger API command.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Inspect the reason given and correct your application.

NON_HEXADECIMAL_OFFSET

Explanation: The supplied offset could not be converted to a binary offset.

Category: InvalidIndependentOfSystemState

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.

Resolution: Ensure the offset is specified as a hexadecimal string.

OFFSET_AFTER_LEDGER_END

Explanation: This rejection is given when a read request uses an offset beyond the current ledger end.

Category: InvalidGivenCurrentSystemStateSeekAfterEnd

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status OUT_OF_RANGE including a detailed error message.

Resolution: Use an offset that is before the ledger end.

OFFSET_OUT_OF_RANGE

Explanation: This rejection is given when a read request uses an offset invalid in the requests' context.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: Inspect the error message and use a valid offset.

PARTICIPANT_PRUNED_DATA_ACCESSED

Explanation: This rejection is given when a read request tries to access pruned data.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.

Resolution: Use an offset that is after the pruning offset.

1.35.9.20 1.3.6.1. ParticipantErrorGroup / LedgerApiErrors / RequestValidation / NotFound

LEDGER_CONFIGURATION_NOT_FOUND

Explanation: The ledger configuration could not be retrieved. This could happen due to incomplete initialization of the participant or due to an internal system error.

Category: InvalidGivenCurrentSystemStateResourceMissing

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.

Resolution: Contact the participant operator.

PACKAGE_NOT_FOUND

Explanation: This rejection is given when a read request tries to access a package which does not exist on the ledger.

Category: InvalidGivenCurrentSystemStateResourceMissing

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.

Resolution: Use a package id pertaining to a package existing on the ledger.

TEMPLATES_OR_INTERFACES_NOT_FOUND

Explanation: The queried template or interface ids do not exist.

Category: InvalidGivenCurrentSystemStateResourceMissing

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.

Resolution: Use valid template or interface ids in your query or ask the participant operator to upload the package containing the necessary interfaces/templates.

TRANSACTION_NOT_FOUND

Explanation: The transaction does not exist or the requesting set of parties are not authorized to fetch it.

Category: InvalidGivenCurrentSystemStateResourceMissing

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.

Resolution: Check the transaction id and verify that the requested transaction is visible to the requesting parties.

1.35.9.21 1.3.7. ParticipantErrorGroup / LedgerApiErrors / WriteServiceRejections

Generic submission rejection errors returned by the backing ledger's write service.

DISPUTED

Deprecation: Corresponds to transaction submission rejections that are not produced anymore. Since: 1.18.0

Explanation: An invalid transaction submission was not detected by the participant.

Category: SystemInternalAssumptionViolated

Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.

Resolution: Contact support.

OUT_OF_QUOTA

Deprecation: Corresponds to transaction submission rejections that are not produced anymore. Since: 1.18.0

Explanation: The Participant node did not have sufficient resource quota to submit the transaction.

Category: ContentionOnSharedResources

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.

Resolution: Inspect the error message and retry after after correcting the underlying issue.

PARTY_NOT_KNOWN_ON_LEDGER

Explanation: One or more informee parties have not been allocated.

Category: InvalidGivenCurrentSystemStateResourceMissing

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.

Resolution: Check that all the informee party identifiers are correct, allocate all the informee parties, request their allocation or wait for them to be allocated before retrying the transaction submission.

SUBMITTER_CANNOT_ACT_VIA_PARTICIPANT

Explanation: A submitting party is not authorized to act through the participant.

Category: InsufficientPermission

Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status PERMISSION_DENIED without any details for security reasons.

Resolution: Contact the participant operator or re-submit with an authorized party.

SUBMITTING_PARTY_NOT_KNOWN_ON_LEDGER

Explanation: The submitting party has not been allocated.

Category: InvalidGivenCurrentSystemStateResourceMissing

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.

Resolution: Check that the party identifier is correct, allocate the submitting party, request its allocation or wait for it to be allocated before retrying the transaction submission.

1.35.9.22 1.3.7.1. ParticipantErrorGroup / LedgerApiErrors / WriteServiceRejections / Internal

Errors that arise from an internal system misbehavior.

INTERNALLY_DUPLICATE_KEYS

Explanation: The participant didn't detect an attempt by the transaction submission to use the same key for two active contracts.

Category: SystemInternalAssumptionViolated

Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.

Resolution: Contact support.

INTERNALLY_INCONSISTENT_KEYS

Explanation: The participant didn't detect an inconsistent key usage in the transaction. Within the transaction, an exercise, fetch or lookupByKey failed because the mapping of *key -> contract ID* was inconsistent with earlier actions.

Category: SystemInternalAssumptionViolated

Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.

Resolution: Contact support.

1.35.10 Error Codes Inventory - Canton

1.35.10.1 1. GrpcErrors

ABORTED_DUE_TO_SHUTDOWN

Explanation: This error is returned when processing of the request was aborted due to the node shutting down.
Resolution: Retry the request against an active and available node.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: ABORTED_DUE_TO_SHUTDOWN

1.35.10.2 2. ParticipantErrorGroup

1.35.10.3 2.1. Errors

ACS_COMMITMENT_INTERNAL_ERROR

Explanation: This error indicates that there was an internal error within the ACS commitment processing.
Resolution: Inspect error message for details.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side.
Scaladocs: ACS_COMMITMENT_INTERNAL_ERROR

1.35.10.4 2.1.1. MismatchError

ACS_COMMITMENT_ALARM

Explanation: The participant has detected that another node is behaving maliciously.
Resolution: Contact support.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: ACS_COMMITMENT_ALARM

ACS_COMMITMENT_MISMATCH

Explanation: This error indicates that a remote participant has sent a commitment over an ACS for a period which does not match the local commitment. This error occurs if a remote participant has manually changed contracts using repair, or due to byzantine behavior, or due to malfunction of the system. The consequence is that the ledger is forked, and some commands that should pass will not.
Resolution: Please contact the other participant in order to check the cause of the mismatch. Either repair the store of this participant or of the counterparty.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: ACS_COMMITMENT_MISMATCH

ACS_MISMATCH_NO_SHARED_CONTRACTS

Explanation: This error indicates that a remote participant has sent a commitment over an ACS for a period, while this participant does not think that there is a shared contract state. This error occurs if a remote participant has manually changed contracts using repair, or due to byzantine behavior, or due to malfunction of the system. The consequence is that the ledger is forked, and some commands that should pass will not.
Resolution: Please contact the other participant in order to check the cause of the mismatch. Either repair the store of this participant or of the counterparty.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: ACS_MISMATCH_NO_SHARED_CONTRACTS

1.35.10.5 2.2. LedgerApiErrors

HEAP_MEMORY_OVER_LIMIT

Explanation: This error happens when the JVM heap memory pool exceeds a pre-configured limit.
Resolution: The following actions can be taken: 1. Review the historical use of heap space by inspecting the metric given in the message. 2. Review the current heap space limits configured

in the rate limiting configuration. 3. Try to space out requests that are likely to require a large amount of memory to process.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: HEAP_MEMORY_OVER_LIMIT

LEDGER_API_INTERNAL_ERROR

Explanation: This error occurs if there was an unexpected error in the Ledger API.
Resolution: Contact support.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: LEDGER_API_INTERNAL_ERROR

MAXIMUM_NUMBER_OF_STREAMS

Explanation: This error happens when the number of concurrent gRPC streaming requests exceeds the configured limit.
Resolution: The following actions can be taken: 1. Review the historical need for concurrent streaming by inspecting the metric given in the message. 2. Review the maximum streams limit configured in the rate limiting configuration. 3. Try to space out streaming requests such that they do not need to run in parallel with each other.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: MAXIMUM_NUMBER_OF_STREAMS

PARTICIPANT_BACKPRESSURE

Explanation: This error occurs when a participant rejects a command due to excessive load. Load can be caused by the following factors: 1. when commands are submitted to the participant through its Ledger API, 2. when the participant receives requests from other participants through a connected domain.
Resolution: Wait a bit and retry, preferably with some backoff factor. If possible, ask other participants to send fewer requests; the domain operator can enforce this by imposing a rate limit.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: PARTICIPANT_BACKPRESSURE

THREADPOOL_OVERLOADED

Explanation: This happens when the rate of submitted gRPC requests requires more CPU or database power than is available.
Resolution: The following actions can be taken: Here the 'queue size' for the threadpool is considered as reported by the executor itself. 1. Review the historical 'queue size' growth by inspecting the metric given in the message. 2. Review the maximum 'queue size' limits configured in the rate limiting configuration. 3. Try to space out requests that are likely to require a lot of CPU or database power.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: THREADPOOL_OVERLOADED

1.35.10.6 2.2.1. CommandExecution

FAILED_TO_DETERMINE_LEDGER_TIME

Explanation: This error occurs if the participant fails to determine the max ledger time of the used contracts. Most likely, this means that one of the contracts is not active anymore which can happen under contention. It can also happen with contract keys.
Resolution: Retry the transaction submission.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: FAILED_TO_DETERMINE_LEDGER_TIME

1.35.10.7 2.2.1.1. Package

ALLOWED_LANGUAGE_VERSIONS

Explanation: This error indicates that the uploaded DAR is based on an unsupported language version.
Resolution: Use a DAR compiled with a language version that this participant supports.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: ALLOWED_LANGUAGE_VERSIONS

PACKAGE_VALIDATION_FAILED

Explanation: This error occurs if a package referred to by a command fails validation. This should not happen as packages are validated when being uploaded.
Resolution: Contact support.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: PACKAGE_VALIDATION_FAILED

1.35.10.8 2.2.1.2. Preprocessing

COMMAND_PREPROCESSING_FAILED

Explanation: This error occurs if a command fails during interpreter pre-processing.
Resolution: Inspect error details and correct your application.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: COMMAND_PREPROCESSING_FAILED

1.35.10.9 2.2.1.3. Interpreter

CONTRACT_DOES_NOT_IMPLEMENT_INTERFACE

Explanation: This error occurs when you try to coerce/use a contract via an interface that it does not implement.
Resolution: Ensure the contract you are calling does implement the interface you are using to do so. Avoid writing LF/low-level interface implementation classes manually.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: CONTRACT_DOES_NOT_IMPLEMENT_INTERFACE

CONTRACT_DOES_NOT_IMPLEMENT_REQUIRING_INTERFACE

Explanation: This error occurs when you try to create/use a contract that does not implement the requiring interfaces of some other interface that it does implement.
Resolution: Ensure you implement all required interfaces correctly, and avoid writing LF/low-level interface implementation classes manually.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: CONTRACT_DOES_NOT_IMPLEMENT_REQUIRING_INTERFACE

CONTRACT_ID_COMPARABILITY

Explanation: This error occurs when you attempt to compare a global and local contract ID of the same discriminator.
Resolution: Avoid constructing contract IDs manually.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: CONTRACT_ID_COMPARABILITY

CONTRACT_ID_IN_CONTRACT_KEY

Explanation: This error occurs when a contract key contains a contract ID, which is illegal for hashing reasons.
Resolution: Ensure your contracts key field cannot contain a contract ID.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: CONTRACT_ID_IN_CONTRACT_KEY

CONTRACT_NOT_ACTIVE

Explanation: This error occurs if an exercise or fetch happens on a transaction-locally consumed contract.
Resolution: This error indicates an application error.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: CONTRACT_NOT_ACTIVE

CREATE_EMPTY_CONTRACT_KEY_MAINTAINERS

Explanation: This error occurs when you try to create a contract that has a key, but with empty maintainers.
Resolution: Check the definition of the contract key's maintainers, and ensure this list won't be empty given your creation arguments.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: CREATE_EMPTY_CONTRACT_KEY_MAINTAINERS

DAML_AUTHORIZATION_ERROR

Explanation: This error occurs if a Daml transaction fails due to an authorization error. An authorization means that the Daml transaction computed a different set of required submitters than you have provided during the submission as *actAs* parties.
Resolution: This error type occurs if there is an application error.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: DAML_AUTHORIZATION_ERROR

DAML_INTERPRETATION_ERROR

Explanation: This error occurs if a Daml transaction fails during interpretation.
Resolution: This error type occurs if there is an application error.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: DAML_INTERPRETATION_ERROR

DAML_INTERPRETER_INVALID_ARGUMENT

Explanation: This error occurs if a Daml transaction fails during interpretation due to an invalid argument.
Resolution: This error type occurs if there is an application error.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: DAML_INTERPRETER_INVALID_ARGUMENT

DISCLOSED_CONTRACT_KEY_HASHING_ERROR

Explanation: This error occurs if a user attempts to provide a key hash for a disclosed contract which we have already cached to be different.
Resolution: Ensure the contract ID and contract payload you have provided in your disclosed contract is correct.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: DISCLOSED_CONTRACT_KEY_HASHING_ERROR

FETCH_EMPTY_CONTRACT_KEY_MAINTAINERS

Explanation: This error occurs when you try to fetch a contract by key, but that key would have empty maintainers.
Resolution: Check the definition of the contract key's maintainers, and ensure this list won't be empty given the contract key you are fetching.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: FETCH_EMPTY_CONTRACT_KEY_MAINTAINERS

INTERPRETATION_DEV_ERROR

Explanation: This error is a catch-all for errors thrown by in-development features, and should never be thrown in production.
Resolution: See the error message for details of the specific in-development feature error. If this is production, avoid using development features.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: INTERPRETATION_DEV_ERROR

INTERPRETATION_USER_ERROR

Explanation: This error occurs when a user calls abort or error on an LF version before native exceptions were introduced.
Resolution: Either remove the call to abort, error or perhaps assert, or ensure you are exercising your contract choice as the author expects.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: INTERPRETATION_USER_ERROR

NON_COMPARABLE_VALUES

Explanation: This error occurs when you attempt to compare two values of different types using the built-in comparison types.
Resolution: Avoid using the low level comparison build, and instead use the Eq class.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: NON_COMPARABLE_VALUES

TEMPLATE_PRECONDITION_VIOLATED

Explanation: This error occurs when a contract's pre-condition (the ensure clause) is violated on contract creation.
Resolution: Ensure the contract argument you are passing into your create doesn't violate the conditions of the contract.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: TEMPLATE_PRECONDITION_VIOLATED

UNHANDLED_EXCEPTION

Explanation: This error occurs when a user throws an error and does not catch it with try-catch.
Resolution: Either your error handling in a choice body is insufficient, or you are using a contract incorrectly.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: UNHANDLED_EXCEPTION

WRONGLY_TYPED_CONTRACT

Explanation: This error occurs when you try to fetch/use a contract in some way with a contract ID that doesn't match the template type on the ledger.
Resolution: Ensure the contract IDs you are using are of the type we expect on the ledger. Avoid unsafely coercing contract IDs.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: WRONGLY_TYPED_CONTRACT

1.35.10.10 2.2.1.3.1. LookupErrors

CONTRACT_KEY_NOT_FOUND

Explanation: This error occurs if the Daml engine interpreter cannot resolve a contract key to an active contract. This can be caused by either the contract key not being known to the participant, or not being known to the submitting parties or the contract representing an already archived key.
Resolution: This error type occurs if there is contention on a contract.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: CONTRACT_KEY_NOT_FOUND

1.35.10.11 2.2.2. AdminServices

CONFIGURATION_ENTRY_REJECTED

Explanation: This rejection is given when a new configuration is rejected.
Resolution: Fetch newest configuration and/or retry.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: CONFIGURATION_ENTRY_REJECTED

INTERNALLY_INVALID_KEY

Explanation: A cryptographic key used by the configured system is not valid
Resolution: Contact support.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: INTERNALLY_INVALID_KEY

PACKAGE_UPLOAD_REJECTED

Explanation: This rejection is given when a package upload is rejected.
Resolution: Refer to the detailed message of the received error.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: PACKAGE_UPLOAD_REJECTED

1.35.10.12 2.2.2.1. PartyManagementServiceErrorGroup

CONCURRENT_PARTY_DETAILS_UPDATE_DETECTED

Explanation: Concurrent updates to a party can be controlled by supplying an update request with a resource version (this is optional). A party's resource version can be obtained by reading the party on the Ledger API. There was attempt to update a party using a stale resource version, indicating that a different process had updated the party earlier.
Resolution: Read this party again to obtain its most recent state and in particular its most recent resource version. Use the obtained information to build and send a new update request.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: CONCURRENT_PARTY_DETAILS_UPDATE_DETECTED

INTERNAL_PARTY_RECORD_ALREADY_EXISTS

Explanation: Each on-ledger party known to this participant node can have a participant's local metadata assigned to it. The local information about a party referred to by this request was found when it should have been not found.
Resolution: This error can indicate a problem with the server's storage or implementation.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: INTERNAL_PARTY_RECORD_ALREADY_EXISTS

INTERNAL_PARTY_RECORD_NOT_FOUND

Explanation: Each on-ledger party known to this participant node can have a participant's local metadata assigned to it. The local information about a party referred to by this request was not found when it should have been found.
Resolution: This error can indicate a problem with the server's storage or implementation.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: INTERNAL_PARTY_RECORD_NOT_FOUND

INVALID_PARTY_DETAILS_UPDATE_REQUEST

Explanation: There was an attempt to update a party using an invalid update request.
Resolution: Inspect the error details for specific information on what made the request invalid. Retry with an adjusted update request.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: INVALID_PARTY_DETAILS_UPDATE_REQUEST

MAX_PARTY_DETAILS_ANNOTATIONS_SIZE_EXCEEDED

Explanation: A party can have at most 256kb worth of annotations in total measured in number of bytes in UTF-8 encoding. There was an attempt to allocate or update a party such that this limit would have been exceeded.
Resolution: Retry with fewer annotations or delete some of the party's existing annotations.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: MAX_PARTY_DETAILS_ANNOTATIONS_SIZE_EXCEEDED

PARTY_NOT_FOUND

Explanation: The party referred to by the request was not found.
Resolution: Check that you are connecting to the right participant node and that the party is spelled correctly.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: PARTY_NOT_FOUND

1.35.10.13 2.2.2.2. UserManagementServiceErrorGroup

CONCURRENT_USER_UPDATE_DETECTED

Explanation: Concurrent updates to a user can be controlled by supplying an update request with a resource version (this is optional). A user's resource version can be obtained by reading the user on the Ledger API. There was attempt to update a user using a stale resource version, indicating that a different process had updated the user earlier.
Resolution: Read this user again to obtain its most recent state and in particular its most recent resource version. Use the obtained information to build and send a new update request.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: CONCURRENT_USER_UPDATE_DETECTED

INVALID_USER_UPDATE_REQUEST

Explanation: There was an attempt to update a user using an invalid update request.
Resolution: Inspect the error details for specific information on what made the request invalid. Retry with an adjusted update request.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: INVALID_USER_UPDATE_REQUEST

MAX_USER_ANNOTATIONS_SIZE_EXCEEDED

Explanation: A user can have at most 256kb worth of annotations in total measured in number of bytes in UTF-8 encoding. There was an attempt to create or update a user such that this limit would have been exceeded.
Resolution: Retry with fewer annotations or delete some of the user's existing annotations.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: MAX_USER_ANNOTATIONS_SIZE_EXCEEDED

TOO_MANY_USER_RIGHTS

Explanation: A user can have only a limited number of user rights. There was an attempt to create a user with too many rights or grant too many rights to a user.
Resolution: Retry with a smaller number of rights or delete some of the already existing rights of this user. Contact the participant operator if the limit is too low.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: TOO_MANY_USER_RIGHTS

USER_ALREADY_EXISTS

Explanation: There already exists a user with the same user-id.
Resolution: Check that you are connecting to the right participant node and the user-id is spelled correctly, or use the user that already exists.
Category: InvalidGivenCurrentSystemStateResourceExists
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ALREADY_EXISTS including a detailed error message.
Scaladocs: USER_ALREADY_EXISTS

USER_NOT_FOUND

Explanation: The user referred to by the request was not found.
Resolution: Check that you are connecting to the right participant node and the user-id is spelled correctly, if yes, create the user.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: USER_NOT_FOUND

1.35.10.14 2.2.2.3. IdentityProviderConfigServiceErrorGroup

IDP_CONFIG_ALREADY_EXISTS

Explanation: There already exists an identity provider configuration with the same identity provider id.
Resolution: Check that you are connecting to the right participant node and the identity provider id is spelled correctly, or use an identity provider that already exists.
Category: InvalidGivenCurrentSystemStateResourceExists
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ALREADY_EXISTS including a detailed error message.
Scaladocs: IDP_CONFIG_ALREADY_EXISTS

IDP_CONFIG_BY_ISSUER_NOT_FOUND

Explanation: The identity provider config referred to by the request was not found.
Resolution: Check that you are connecting to the right participant node and the identity provider config is spelled correctly, or create the configuration.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: IDP_CONFIG_BY_ISSUER_NOT_FOUND

IDP_CONFIG_ISSUER_ALREADY_EXISTS

Explanation: There already exists an identity provider configuration with the same issuer.
Resolution: Check that you are connecting to the right participant node and the identity provider id is spelled correctly, or use an identity provider that already exists.
Category: InvalidGivenCurrentSystemStateResourceExists
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ALREADY_EXISTS including a detailed error message.
Scaladocs: IDP_CONFIG_ISSUER_ALREADY_EXISTS

IDP_CONFIG_NOT_FOUND

Explanation: The identity provider config referred to by the request was not found.
Resolution: Check that you are connecting to the right participant node and the identity provider config is spelled correctly, or create the configuration.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: IDP_CONFIG_NOT_FOUND

INVALID_IDENTITY_PROVIDER_UPDATE_REQUEST

Explanation: There was an attempt to update an identity provider config using an invalid update request.
Resolution: Inspect the error details for specific information on what made the request invalid. Retry with an adjusted update request.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: INVALID_IDENTITY_PROVIDER_UPDATE_REQUEST

TOO_MANY_IDENTITY_PROVIDER_CONFIGS

Explanation: A system can have only a limited number of identity provider configurations. There was an attempt to create an identity provider configuration.
Resolution: Delete some of the already existing identity provider configurations. Contact the participant operator if the limit is too low.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: TOO_MANY_IDENTITY_PROVIDER_CONFIGS

1.35.10.15 2.2.3. ConsistencyErrors

CONTRACT_NOT_FOUND

Explanation: This error occurs if the Daml engine can not find a referenced contract. This can be caused by either the contract not being known to the participant, or not being known to the submitting parties or already being archived.
Resolution: This error type occurs if there is contention on a contract.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: CONTRACT_NOT_FOUND

DISCLOSED_CONTRACT_INVALID

Explanation: This error occurs if the disclosed payload or metadata of one of the contracts does not match the actual payload or metadata of the contract.
Resolution: Re-submit the command using valid disclosed contract payload and metadata.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: DISCLOSED_CONTRACT_INVALID

DUPLICATE_COMMAND

Explanation: A command with the given command id has already been successfully processed.
Resolution: The correct resolution depends on the use case. If the error received pertains to a submission retried due to a timeout, do nothing, as the previous command has already been accepted. If the intent is to submit a new command, re-submit using a distinct command id.
Category: InvalidGivenCurrentSystemStateResourceExists
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ALREADY_EXISTS including a detailed error message.
Scaladocs: DUPLICATE_COMMAND

DUPLICATE_CONTRACT_KEY

Explanation: This error signals that within the transaction we got to a point where two contracts with the same key were active.
Resolution: This error indicates an application error.
Category: InvalidGivenCurrentSystemStateResourceExists
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ALREADY_EXISTS including a detailed error message.
Scaladocs: DUPLICATE_CONTRACT_KEY

INCONSISTENT

Explanation: At least one input has been altered by a concurrent transaction submission.
Resolution: The correct resolution depends on the business flow, for example it may be possible to proceed without an archived contract as an input, or the transaction submission may be retried to load the up-to-date value of a contract key.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: INCONSISTENT

INCONSISTENT_CONTRACTS

Explanation: An input contract has been archived by a concurrent transaction submission.
Resolution: The correct resolution depends on the business flow, for example it may be possible to proceed without the archived contract as an input, or a different contract could be used.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: INCONSISTENT_CONTRACTS

INCONSISTENT_CONTRACT_KEY

Explanation: An input contract key was re-assigned to a different contract by a concurrent transaction submission.
Resolution: Retry the transaction submission.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: INCONSISTENT_CONTRACT_KEY

INVALID_LEDGER_TIME

Explanation: The ledger time of the submission violated some constraint on the ledger time.
Resolution: Retry the transaction submission.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: INVALID_LEDGER_TIME

SUBMISSION_ALREADY_IN_FLIGHT

Explanation: Another command submission with the same change ID (application ID, command ID, actAs) is already being processed.
Resolution: Listen to the command completion stream until a completion for the in-flight command submission is published. Alternatively, resubmit the command. If the in-flight submission has finished successfully by then, this will return more detailed information about the earlier one. If the in-flight submission has failed by then, the resubmission will attempt to record the new transaction on the ledger.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: SUBMISSION_ALREADY_IN_FLIGHT

1.35.10.16 2.2.4. PackageServiceError

DAR_NOT_SELF_CONSISTENT

Explanation: This error indicates that the uploaded Dar is broken because it is missing internal dependencies.
Resolution: Contact the supplier of the Dar.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: DAR_NOT_SELF_CONSISTENT

DAR_VALIDATION_ERROR

Explanation: This error indicates that the validation of the uploaded dar failed.
Resolution: Inspect the error message and contact support.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: DAR_VALIDATION_ERROR

PACKAGE_SERVICE_INTERNAL_ERROR

Explanation: This error indicates an internal issue within the package service.
Resolution: Inspect the error message and contact support.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: PACKAGE_SERVICE_INTERNAL_ERROR

SHUTDOWN_INTERRUPTED_PACKAGE_VETTING

Explanation: Package vetting has been aborted because the participant is shutting down.
Resolution: Re-submit the vetting request when the participant node is available again.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: SHUTDOWN_INTERRUPTED_PACKAGE_VETTING

1.35.10.17 2.2.4.1. Reading

DAR_PARSE_ERROR

Explanation: This error indicates that the content of the Dar file could not be parsed successfully.
Resolution: Inspect the error message and contact support.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: DAR_PARSE_ERROR

INVALID_DAR

Explanation: This error indicates that the supplied dar file was invalid.
Resolution: Inspect the error message for details and contact support.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: INVALID_DAR

INVALID_DAR_FILE_NAME

Explanation: This error indicates that the supplied dar file name did not meet the requirements to be stored in the persistence store.
Resolution: Inspect error message for details and change the file name accordingly
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: INVALID_DAR_FILE_NAME

INVALID_LEGACY_DAR

Explanation: This error indicates that the supplied zipped dar is an unsupported legacy Dar.
Resolution: Please use a more recent dar version.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: INVALID_LEGACY_DAR

INVALID_ZIP_ENTRY

Explanation: This error indicates that the supplied zipped dar file was invalid.
Resolution: Inspect the error message for details and contact support.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: INVALID_ZIP_ENTRY

ZIP_BOMB

Explanation: This error indicates that the supplied zipped dar is regarded as zip-bomb.
Resolution: Inspect the dar and contact support.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: ZIP_BOMB

1.35.10.18 2.2.5. WriteServiceRejections

DISPUTED

Explanation: An invalid transaction submission was not detected by the participant.
Resolution: Contact support.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: DISPUTED

OUT_OF_QUOTA

Explanation: The Participant node did not have sufficient resource quota to submit the transaction.
Resolution: Inspect the error message and retry after after correcting the underlying issue.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: OUT_OF_QUOTA

PARTY_NOT_KNOWN_ON_LEDGER

Explanation: One or more informee parties have not been allocated.
Resolution: Check that all the informee party identifiers are correct, allocate all the informee parties, request their allocation or wait for them to be allocated before retrying the transaction submission.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: PARTY_NOT_KNOWN_ON_LEDGER

SUBMITTER_CANNOT_ACT_VIA_PARTICIPANT

Explanation: A submitting party is not authorized to act through the participant.
Resolution: Contact the participant operator or re-submit with an authorized party.
Category: InsufficientPermission
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status PERMISSION_DENIED without any details for security reasons.
Scaladocs: SUBMITTER_CANNOT_ACT_VIA_PARTICIPANT

SUBMITTING_PARTY_NOT_KNOWN_ON_LEDGER

Explanation: The submitting party has not been allocated.
Resolution: Check that the party identifier is correct, allocate the submitting party, request its allocation or wait for it to be allocated before retrying the transaction submission.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: SUBMITTING_PARTY_NOT_KNOWN_ON_LEDGER

1.35.10.19 2.2.5.1. Internal

INTERNALLY_DUPLICATE_KEYS

Explanation: The participant didn't detect an attempt by the transaction submission to use the same key for two active contracts.
Resolution: Contact support.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: INTERNALLY_DUPLICATE_KEYS

INTERNALLY_INCONSISTENT_KEYS

Explanation: The participant didn't detect an inconsistent key usage in the transaction. Within the transaction, an exercise, fetch or lookupByKey failed because the mapping of *key -> contract ID* was inconsistent with earlier actions.
Resolution: Contact support.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: INTERNALLY_INCONSISTENT_KEYS

1.35.10.20 2.2.6. AuthorizationChecks

INTERNAL_AUTHORIZATION_ERROR

Explanation: An internal system authorization error occurred.
Resolution: Contact the participant operator.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: INTERNAL_AUTHORIZATION_ERROR

PERMISSION_DENIED

Explanation: This rejection is given if the supplied authorization token is not sufficient for the intended command. The exact reason is logged on the participant, but not given to the user for security reasons.
Resolution: Inspect your command and your token or ask your participant operator for an explanation why this command failed.
Category: InsufficientPermission
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status PERMISSION_DENIED without any details for security reasons.
Scaladocs: PERMISSION_DENIED

STALE_STREAM_AUTHORIZATION

Explanation: The stream was aborted because the authenticated user's rights changed, and the user might thus no longer be authorized to this stream.
Resolution: The application should automatically retry fetching the stream. It will either succeed, or fail with an explicit denial of authentication or permission.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: STALE_STREAM_AUTHORIZATION

UNAUTHENTICATED

Explanation: This rejection is given if the submitted command does not contain a JWT token on a participant enforcing JWT authentication.
Resolution: Ask your participant operator to provide you with an appropriate JWT token.
Category: AuthInterceptorInvalidAuthenticationCredentials
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status UNAUTHENTICATED without any details for security reasons.
Scaladocs: UNAUTHENTICATED

1.35.10.21 2.2.7. RequestValidation

INVALID_ARGUMENT

Explanation: This error is emitted when a submitted ledger API command contains an invalid argument.
Resolution: Inspect the reason given and correct your application.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: INVALID_ARGUMENT

INVALID_DEDUPLICATION_PERIOD

Explanation: This error is emitted when a submitted ledger API command specifies an invalid deduplication period.
Resolution: Inspect the error message, adjust the value of the deduplication period or ask the participant operator to increase the maximum deduplication period.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: INVALID_DEDUPLICATION_PERIOD

INVALID_FIELD

Explanation: This error is emitted when a submitted ledger API command contains a field value that cannot be understood.
Resolution: Inspect the reason given and correct your application.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: INVALID_FIELD

LEDGER_ID_MISMATCH

Explanation: Every ledger API command contains a ledger-id which is verified against the running ledger. This error indicates that the provided ledger-id does not match the expected one.
Resolution: Ensure that your application is correctly configured to use the correct ledger.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: LEDGER_ID_MISMATCH

MISSING_FIELD

Explanation: This error is emitted when a mandatory field is not set in a submitted ledger API command.
Resolution: Inspect the reason given and correct your application.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: MISSING_FIELD

NON_HEXADECIMAL_OFFSET

Explanation: The supplied offset could not be converted to a binary offset.
Resolution: Ensure the offset is specified as a hexadecimal string.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: NON_HEXADECIMAL_OFFSET

OFFSET_AFTER_LEDGER_END

Explanation: This rejection is given when a read request uses an offset beyond the current ledger end.
Resolution: Use an offset that is before the ledger end.
Category: InvalidGivenCurrentSystemStateSeekAfterEnd
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status OUT_OF_RANGE including a detailed error message.
Scaladocs: OFFSET_AFTER_LEDGER_END

OFFSET_OUT_OF_RANGE

Explanation: This rejection is given when a read request uses an offset invalid in the requests' context.
Resolution: Inspect the error message and use a valid offset.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: OFFSET_OUT_OF_RANGE

PARTICIPANT_PRUNED_DATA_ACCESSED

Explanation: This rejection is given when a read request tries to access pruned data.
Resolution: Use an offset that is after the pruning offset.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: PARTICIPANT_PRUNED_DATA_ACCESSED

1.35.10.22 2.2.7.1. NotFound

LEDGER_CONFIGURATION_NOT_FOUND

Explanation: The ledger configuration could not be retrieved. This could happen due to incomplete initialization of the participant or due to an internal system error.
Resolution: Contact the participant operator.
Category: InvalidGivenCurrentSystemStateResourceMissing

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: LEDGER_CONFIGURATION_NOT_FOUND

PACKAGE_NOT_FOUND

Explanation: This rejection is given when a read request tries to access a package which does not exist on the ledger.
Resolution: Use a package id pertaining to a package existing on the ledger.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: PACKAGE_NOT_FOUND

TEMPLATES_OR_INTERFACES_NOT_FOUND

Explanation: The queried template or interface ids do not exist.
Resolution: Use valid template or interface ids in your query or ask the participant operator to upload the package containing the necessary interfaces/templates.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: TEMPLATES_OR_INTERFACES_NOT_FOUND

TRANSACTION_NOT_FOUND

Explanation: The transaction does not exist or the requesting set of parties are not authorized to fetch it.
Resolution: Check the transaction id and verify that the requested transaction is visible to the requesting parties.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: TRANSACTION_NOT_FOUND

1.35.10.23 2.2.7.2. Error

STATE_REQUEST_VALIDATION

Explanation: This error results if a state request failed validation.
Resolution: Check the request.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: STATE_REQUEST_VALIDATION

1.35.10.24 2.3. TransactionErrorGroup

1.35.10.25 2.3.1. TransactionRoutingError

AUTOMATIC_TRANSFER_FOR_TRANSACTION_FAILED

Explanation: This error indicates that the automated transfer could not succeed, as the current topology does not allow the transfer to complete, mostly due to lack of confirmation permissions of the involved parties.
Resolution: Inspect the message and your topology and ensure appropriate permissions exist.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: AUTOMATIC_TRANSFER_FOR_TRANSACTION_FAILED

ROUTING_INTERNAL_ERROR

Explanation: This error indicates an internal error in the Canton domain router.
Resolution: Please contact support.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: ROUTING_INTERNAL_ERROR

UNABLE_TO_GET_TOPOLOGY_SNAPSHOT

Explanation: This error indicates that topology information could not be queried.
Resolution: Check that the participant is connected to the domain.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: UNABLE_TO_GET_TOPOLOGY_SNAPSHOT

1.35.10.26 2.3.1.1. MalformedInputErrors

DISCLOSED_CONTRACT_AUTHENTICATION_FAILED

Explanation: A provided disclosed contract could not be authenticated against the provided contract id.
Resolution: Ensure that disclosed contracts provided with command submission match the original contract creation content as sourced from the Ledger API. If the problem persists, contact the participant operator.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: DISCLOSED_CONTRACT_AUTHENTICATION_FAILED

INVALID_DISCLOSED_CONTRACT

Explanation: A provided disclosed contract could not be processed.
Resolution: Ensure that disclosed contracts provided with command submission have an authenticated contract id (i.e. have been created in participant nodes running Canton protocol version 4 or higher) and match the original contract creation format and content as sourced from the Ledger API. If the problem persists, contact the participant operator.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: INVALID_DISCLOSED_CONTRACT

INVALID_DOMAIN_ALIAS

Explanation: The WorkflowID defined in the transaction metadata is not a valid domain alias.
Resolution: Check that the workflow ID (if specified) corresponds to a valid domain alias. A typical rejection reason is a too-long domain alias.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: INVALID_DOMAIN_ALIAS

INVALID_DOMAIN_ID

Explanation: The WorkflowID defined in the transaction metadata contains an invalid domain id.
Resolution: Check that the workflow ID (if specified) corresponds to a valid domain ID after the `domain-id:` marker string.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: INVALID_DOMAIN_ID

INVALID_PARTY_IDENTIFIER

Explanation: The given party identifier is not a valid Canton party identifier.
Resolution: Ensure that your commands only refer to correct and valid Canton party identifiers of parties that are properly enabled on the system
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: INVALID_PARTY_IDENTIFIER

INVALID_SUBMITTER

Explanation: The party defined as a submitter can not be parsed into a valid Canton party.
Resolution: Check that you only use correctly setup party names in your application.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: INVALID_SUBMITTER

1.35.10.27 2.3.1.2. TopologyErrors

INFORMEES_NOT_ACTIVE

Explanation: This error indicates that the informees are known, but there is no connected domain on which all the informees are hosted.
Resolution: Ensure that there is such a domain, as Canton requires a domain where all informees are present.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: INFORMEES_NOT_ACTIVE

NOT_CONNECTED_TO_ALL_CONTRACT_DOMAINS

Explanation: This error indicates that the transaction is referring to contracts on domains to which this participant is currently not connected.
Resolution: Check the status of your domain connections.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: NOT_CONNECTED_TO_ALL_CONTRACT_DOMAINS

NO_COMMON_DOMAIN

Explanation: This error indicates that there is no common domain to which all submitters can submit and all informees are connected.
Resolution: Check that your participant node is connected to all domains you expect and check that the parties are hosted on these domains as you expect them to be.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: NO_COMMON_DOMAIN

NO_DOMAIN_FOR_SUBMISSION

Explanation: This error indicates that no valid domain was found for submission.
Resolution: Check the status of your domain connections, that packages are vetted and that you are connected to domains running recent protocol versions.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: NO_DOMAIN_FOR_SUBMISSION

NO_DOMAIN_ON_WHICH_ALL_SUBMITTERS_CAN_SUBMIT

Explanation: This error indicates that a transaction has been sent where the system can not find any active + domain on which this participant can submit in the name of the given set of submitters.
Resolution: Ensure that you are connected to a domain where this participant has submission rights. Check that you are actually connected to the domains you expect to be connected and check that your participant node has the submission permission for each submitting party.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: NO_DOMAIN_ON_WHICH_ALL_SUBMITTERS_CAN_SUBMIT

SUBMITTERS_NOT_ACTIVE

Explanation: This error indicates that the submitters are known, but there is no connected domain on which all the submitters are hosted.
Resolution: Ensure that there is such a domain, as Canton requires a domain where all submitters are present.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: SUBMITTERS_NOT_ACTIVE

SUBMITTER_ALWAYS_STAKEHOLDER

Explanation: This error indicates that the transaction requires contract transfers for which the submitter must be a stakeholder.
Resolution: Check that your participant node is connected to all domains you expect and check that the parties are hosted on these domains as you expect them to be.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: SUBMITTER_ALWAYS_STAKEHOLDER

UNKNOWN_CONTRACT_DOMAINS

Explanation: This error indicates that the transaction is referring to contracts whose domain is not currently known.
Resolution: Ensure all transfer operations on contracts used by the transaction have completed.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: UNKNOWN_CONTRACT_DOMAINS

UNKNOWN_INFORMEES

Explanation: This error indicates that the transaction is referring to some informees that are not known on any connected domain.
Resolution: Check the list of submitted informees and check if your participant is connected to the domains you are expecting it to be.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: UNKNOWN_INFORMEES

UNKNOWN_SUBMITTERS

Explanation: This error indicates that the transaction is referring to some submitters that are not known on any connected domain.
Resolution: Check the list of provided submitters and check if your participant is connected to the domains you are expecting it to be.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: UNKNOWN_SUBMITTERS

1.35.10.28 2.3.1.3. ConfigurationErrors

INVALID_PRESCRIBED_DOMAIN_ID

Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: INVALID_PRESCRIBED_DOMAIN_ID

MULTI_DOMAIN_SUPPORT_NOT_ENABLED

Explanation: This error indicates that a transaction has been submitted that requires multi-domain support. Multi-domain support is a preview feature that needs to be enabled explicitly by configuration.
Resolution: Set canton.features.enable-preview-commands = yes
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: MULTI_DOMAIN_SUPPORT_NOT_ENABLED

SUBMISSION_DOMAIN_NOT_READY

Explanation: This error indicates that the transaction should be submitted to a domain which is not connected or not configured.
Resolution: Ensure that the domain specified in the workflowId is correctly connected.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: SUBMISSION_DOMAIN_NOT_READY

1.35.10.29 2.3.2. SubmissionErrors

CHOSEN_MEDIATOR_IS_INACTIVE

Explanation: The mediator chosen for the transaction got deactivated before the request was sequenced.
Resolution: Resubmit.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: CHOSEN_MEDIATOR_IS_INACTIVE

CONTRACT_AUTHENTICATION_FAILED

Explanation: At least one of the transaction's input contracts could not be authenticated.
Resolution: Retry the submission with correctly authenticated contracts.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: CONTRACT_AUTHENTICATION_FAILED

DOMAIN_BACKPRESSURE

Explanation: This error occurs when the sequencer refuses to accept a command due to back-pressure.
Resolution: Wait a bit and retry, preferably with some backoff factor.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: DOMAIN_BACKPRESSURE

DOMAIN_WITHOUT_MEDIATOR

Explanation: The participant routed the transaction to a domain without an active mediator.
Resolution: Add a mediator to the domain.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: DOMAIN_WITHOUT_MEDIATOR

MALFORMED_REQUEST

Explanation: This error has not yet been properly categorised into sub-error codes.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: MALFORMED_REQUEST

NOT_SEQUENCED_TIMEOUT

Explanation: This error occurs when the transaction was not sequenced within the pre-defined max-sequencing time and has therefore timed out. The max-sequencing time is derived from the transaction's ledger time via the ledger time model skews.
Resolution: Resubmit if the delay is caused by high load. If the command requires substantial processing on the participant, specify a higher minimum ledger time with the command submission so that a higher max sequencing time is derived.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: NOT_SEQUENCED_TIMEOUT

PACKAGE_NOT_VETTED_BY_RECIPIENTS

Explanation: This error occurs if a transaction was submitted referring to a package that a receiving participant has not vetted. Any transaction view can only refer to packages that have explicitly been approved by the receiving participants.
Resolution: Ensure that the receiving participant uploads and vets the respective package.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: PACKAGE_NOT_VETTED_BY_RECIPIENTS

PARTICIPANT_OVERLOADED

Explanation: The participant has rejected all incoming commands during a configurable grace period.
Resolution: Configure more restrictive resource limits (enterprise only). Change applications to submit commands at a lower rate. Configure a higher value for *myParticipant.parameters.warnIfOverloadedFor*.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level WARN on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: PARTICIPANT_OVERLOADED

SEQUENCER_DELIVER_ERROR

Explanation: This error occurs when the domain refused to sequence the given message.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: SEQUENCER_DELIVER_ERROR

SEQUENCER_REQUEST_FAILED

Explanation: This error occurs when the command cannot be sent to the domain.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: SEQUENCER_REQUEST_FAILED

SUBMISSION_DURING_SHUTDOWN

Explanation: This error occurs when a command is submitted while the system is performing a shutdown.
Resolution: Assuming that the participant will restart or failover eventually, retry in a couple of seconds.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: SUBMISSION_DURING_SHUTDOWN

1.35.10.30 2.3.3. SyncServiceInjectionError

COMMAND_INJECTION_FAILURE

Explanation: This errors occurs if an internal error results in an exception.
Resolution: Contact support.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: COMMAND_INJECTION_FAILURE

NODE_IS_PASSIVE_REPLICA

Explanation: This error results if a command is submitted to the passive replica.
Resolution: Send the command to the active replica.
Category: TransientServerFailure
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status UNAVAILABLE including a detailed error message.
Scaladocs: NODE_IS_PASSIVE_REPLICA

NOT_CONNECTED_TO_ANY_DOMAIN

Explanation: This errors results if a command is submitted to a participant that is not connected to any domain.
Resolution: Connect your participant to the domain where the given parties are hosted.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: NOT_CONNECTED_TO_ANY_DOMAIN

1.35.10.31 2.3.4. LocalReject

1.35.10.32 2.3.4.1. MalformedRejects

LOCAL_VERDICT_BAD_ROOT_HASH_MESSAGES

Explanation: This rejection is made by a participant if a transaction does not contain valid root hash messages.
Resolution: This indicates a race condition due to a in-flight topology change, or malicious or faulty behaviour.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: LOCAL_VERDICT_BAD_ROOT_HASH_MESSAGES

LOCAL_VERDICT_CREATES_EXISTING_CONTRACTS

Explanation: This error indicates that the transaction would create already existing contracts.
Resolution: This error indicates either faulty or malicious behaviour.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: LOCAL_VERDICT_CREATES_EXISTING_CONTRACTS

LOCAL_VERDICT_FAILED_MODEL_CONFORMANCE_CHECK

Explanation: This rejection is made by a participant if a transaction fails a model conformance check.
Resolution: This indicates either malicious or faulty behaviour.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: LOCAL_VERDICT_FAILED_MODEL_CONFORMANCE_CHECK

LOCAL_VERDICT_MALFORMED_PAYLOAD

Explanation: This rejection is made by a participant if a view of the transaction is malformed.
Resolution: This indicates either malicious or faulty behaviour.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: LOCAL_VERDICT_MALFORMED_PAYLOAD

LOCAL_VERDICT_MALFORMED_REQUEST

Explanation: This rejection is made by a participant if a request is malformed.
Resolution: Please contact support.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: LOCAL_VERDICT_MALFORMED_REQUEST

1.35.10.33 2.3.4.2. ConsistencyRejections

LOCAL_VERDICT_DUPLICATE_KEY

Explanation: If the participant provides unique contract key support, this error will indicate that a transaction would create a unique key which already exists.
Resolution: It depends on your use case and application whether and when retrying makes sense or not.
Category: InvalidGivenCurrentSystemStateResourceExists
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ALREADY_EXISTS including a detailed error message.
Scaladocs: LOCAL_VERDICT_DUPLICATE_KEY

LOCAL_VERDICT_INACTIVE_CONTRACTS

Explanation: The transaction is referring to contracts that have either been previously archived, transferred to another domain, or do not exist.
Resolution: Inspect your contract state and try a different transaction.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: LOCAL_VERDICT_INACTIVE_CONTRACTS

LOCAL_VERDICT_INCONSISTENT_KEY

Explanation: If the participant provides unique contract key support, this error will indicate that a transaction expected a key to be unallocated, but a contract for the key already exists.
Resolution: It depends on your use case and application whether and when retrying makes sense or not.
Category: InvalidGivenCurrentSystemStateResourceExists
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ALREADY_EXISTS including a detailed error message.
Scaladocs: LOCAL_VERDICT_INCONSISTENT_KEY

LOCAL_VERDICT_LOCKED_CONTRACTS

Explanation: The transaction is referring to locked contracts which are in the process of being created, transferred, or archived by another transaction. If the other transaction fails, this transaction could be successfully retried.
Resolution: Retry the transaction
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: LOCAL_VERDICT_LOCKED_CONTRACTS

LOCAL_VERDICT_LOCKED_KEYS

Explanation: The transaction is referring to locked keys which are in the process of being modified by another transaction.
Resolution: Retry the transaction
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: LOCAL_VERDICT_LOCKED_KEYS

1.35.10.34 2.3.4.3. TimeRejects

LOCAL_VERDICT_LEDGER_TIME_OUT_OF_BOUND

Explanation: This error is thrown if the ledger time and the record time differ more than permitted. This can happen in an overloaded system due to high latencies or for transactions with long interpretation times.
Resolution: For long-running transactions, specify a ledger time with the command submission or adjust the dynamic domain parameter ledgerTimeRecordTimeTolerance (and possibly the participant and mediator reaction timeout). For short-running transactions, simply retry.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: LOCAL_VERDICT_LEDGER_TIME_OUT_OF_BOUND

LOCAL_VERDICT_SUBMISSION_TIME_OUT_OF_BOUND

Explanation: This error is thrown if the submission time and the record time differ more than permitted. This can happen in an overloaded system due to high latencies or for transactions with long interpretation times.
Resolution: For long-running transactions, adjust the ledger time bounds used with the command submission. For short-running transactions, simply retry.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: LOCAL_VERDICT_SUBMISSION_TIME_OUT_OF_BOUND

LOCAL_VERDICT_TIMEOUT

Explanation: This rejection is sent if the participant locally determined a timeout.
Resolution: In the first instance, resubmit your transaction. If the rejection still appears spuriously, consider increasing the *participantResponseTimeout* or *mediatorReactionTimeout* values in the *DynamicDomainParameters*. If the rejection appears unrelated to timeout settings, validate that all other Canton components which take part in the transaction also function correctly and that, e.g., messages are not stuck at the sequencer.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level WARN on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: LOCAL_VERDICT_TIMEOUT

1.35.10.35 2.3.4.4. TransferInRejects

TRANSFER_IN_ALREADY_COMPLETED

Explanation: This rejection is emitted by a participant if a transfer-in has already been completed.
Category: InvalidGivenCurrentSystemStateResourceExists
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ALREADY_EXISTS including a detailed error message.
Scaladocs: TRANSFER_IN_ALREADY_COMPLETED

TRANSFER_IN_CONTRACT_ALREADY_ACTIVE

Explanation: This rejection is emitted by a participant if a transfer-in has already been made by another entity.
Category: InvalidGivenCurrentSystemStateResourceExists
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ALREADY_EXISTS including a detailed error message.
Scaladocs: TRANSFER_IN_CONTRACT_ALREADY_ACTIVE

TRANSFER_IN_CONTRACT_ALREADY_ARCHIVED

Explanation: This rejection is emitted by a participant if a transfer would be invoked on an already archived contract.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: TRANSFER_IN_CONTRACT_ALREADY_ARCHIVED

TRANSFER_IN_CONTRACT_IS_LOCKED

Explanation: This rejection is emitted by a participant if a transfer-in is referring to an already locked contract.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: TRANSFER_IN_CONTRACT_IS_LOCKED

1.35.10.36 2.3.4.5. TransferOutRejects

TRANSFER_OUT_ACTIVENESS_CHECK_FAILED

Explanation: Activeness check failed for transfer out submission. This rejection occurs if the contract to be transferred has already been transferred or is currently locked (due to a competing transaction) on domain.
Resolution: Depending on your use-case and your expectation, retry the transaction.
Category: InvalidGivenCurrentSystemStateResourceMissing

Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: TRANSFER_OUT_ACTIVENESS_CHECK_FAILED

1.35.10.37 2.3.5. CommandDeduplicationError

MALFORMED_DEDUPLICATION_OFFSET

Explanation: The specified deduplication offset is syntactically malformed.
Resolution: Use a deduplication offset that was produced by this participant node.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: MALFORMED_DEDUPLICATION_OFFSET

1.35.10.38 2.3.6. Error

TRANSFER_COMMAND_VALIDATION

Explanation: This error results if a transfer command fails initial validation.
Resolution: Check connection to the domain and the topology.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: TRANSFER_COMMAND_VALIDATION

1.35.10.39 2.4. SyncServiceError

PARTY_ALLOCATION_WITHOUT_CONNECTED_DOMAIN

Explanation: The participant is not connected to a domain and can therefore not allocate a party because the party notification is configured as `party-notification.type = via-domain`.
Resolution: Connect the participant to a domain first or change the participant's party notification config to `eager`.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: PARTY_ALLOCATION_WITHOUT_CONNECTED_DOMAIN

SYNC_SERVICE_ALARM

Explanation: The participant has detected that another node is behaving maliciously.
Resolution: Contact support.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: SYNC_SERVICE_ALARM

SYNC_SERVICE_ALREADY_ADDED

Explanation: This error results on an attempt to register a new domain under an alias already in use.
Category: InvalidGivenCurrentSystemStateResourceExists
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ALREADY_EXISTS including a detailed error message.
Scaladocs: SYNC_SERVICE_ALREADY_ADDED

SYNC_SERVICE_DOMAIN_BECAME_PASSIVE

Explanation: This error is logged when a sync domain is disconnected because the participant became passive.
Resolution: Fail over to the active participant replica.
Category: TransientServerFailure
Conveyance: This error is logged with log-level WARN on the server side and exposed on the API with grpc-status UNAVAILABLE including a detailed error message.
Scaladocs: SYNC_SERVICE_DOMAIN_BECAME_PASSIVE

SYNC_SERVICE_DOMAIN_DISABLED_US

Explanation: This error is logged when the synchronization service shuts down because the remote domain has disabled this participant.
Resolution: Contact the domain operator and inquire why you have been booted out.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level WARN on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: SYNC_SERVICE_DOMAIN_DISABLED_US

SYNC_SERVICE_DOMAIN_DISCONNECTED

Explanation: This error is logged when a sync domain is unexpectedly disconnected from the Canton sync service (after having previously been connected)
Resolution: Please contact support and provide the failure reason.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: SYNC_SERVICE_DOMAIN_DISCONNECTED

SYNC_SERVICE_DOMAIN_MUST_BE_OFFLINE

Explanation: This error is emitted when an operation is attempted such as repair that requires the domain connection to be disconnected and clean.
Resolution: Disconnect the domain before attempting the command.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: SYNC_SERVICE_DOMAIN_MUST_BE_OFFLINE

SYNC_SERVICE_DOMAIN_STATUS_NOT_ACTIVE

Explanation: This error is logged when a sync domain has a non-active status.
Resolution: If you attempt to connect to a domain that has either been migrated off or has a pending migration, this error will be emitted. Please complete the migration before attempting to connect to it.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: SYNC_SERVICE_DOMAIN_STATUS_NOT_ACTIVE

SYNC_SERVICE_INTERNAL_ERROR

Explanation: This error indicates an internal issue.
Resolution: Please contact support and provide the failure reason.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: SYNC_SERVICE_INTERNAL_ERROR

SYNC_SERVICE_UNKNOWN_DOMAIN

Explanation: This error results if a domain connectivity command is referring to a domain alias that has not been registered.
Resolution: Please confirm the domain alias is correct, or configure the domain before (re)connecting.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: SYNC_SERVICE_UNKNOWN_DOMAIN

1.35.10.40 2.4.1. SyncDomainMigrationError

BROKEN_DOMAIN_MIGRATION

Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: BROKEN_DOMAIN_MIGRATION

INVALID_DOMAIN_MIGRATION_REQUEST

Explanation: This error results when invalid arguments are passed to the migration command.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: INVALID_DOMAIN_MIGRATION_REQUEST

1.35.10.41 2.4.2. DomainRegistryError

DOMAIN_REGISTRY_INTERNAL_ERROR

Explanation: This error indicates that there has been an internal error noticed by Canton.
Resolution: Contact support and provide the failure reason.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: DOMAIN_REGISTRY_INTERNAL_ERROR

TOPOLOGY_CONVERSION_ERROR

Explanation: This error indicates that there was an error converting topology transactions during connecting to a domain.
Resolution: Contact the operator of the topology management for this node.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: TOPOLOGY_CONVERSION_ERROR

1.35.10.42 2.4.2.1. ConfigurationErrors

CANNOT_ISSUE_DOMAIN_TRUST_CERTIFICATE

Explanation: This error indicates that the participant can not issue a domain trust certificate. Such a certificate is necessary to become active on a domain. Therefore, it must be present in the authorized store of the participant topology manager.
Resolution: Manually upload a valid domain trust certificate for the given domain or upload the necessary certificates such that participant can issue such certificates automatically.

Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: CANNOT_ISSUE_DOMAIN_TRUST_CERTIFICATE

DOMAIN_PARAMETERS_CHANGED

Explanation: Error indicating that the domain parameters have been changed, while this isn't supported yet.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level WARN on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: DOMAIN_PARAMETERS_CHANGED

INCOMPATIBLE_UNIQUE_CONTRACT_KEYS_MODE

Explanation: This error indicates that the domain this participant is trying to connect to is a domain where unique contract keys are supported, while this participant is already connected to other domains. Multiple domains and unique contract keys are mutually exclusive features.
Resolution: Use isolated participants for domains that require unique keys.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: INCOMPATIBLE_UNIQUE_CONTRACT_KEYS_MODE

INVALID_DOMAIN_CONNECTION

Explanation: This error indicates there is a validation error with the configured connections for the domain
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: INVALID_DOMAIN_CONNECTION

MISCONFIGURED_STATIC_DOMAIN_PARAMETERS

Explanation: This error indicates that the participant is configured to connect to multiple domain sequencers but their static domain parameters are different from other sequencers.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: MISCONFIGURED_STATIC_DOMAIN_PARAMETERS

SEQUENCERS_FROM_DIFFERENT_DOMAINS_ARE_CONFIGURED

Explanation: This error indicates that the participant is configured to connect to multiple domain sequencers from different domains.
Resolution: Carefully verify the connection settings.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: SEQUENCERS_FROM_DIFFERENT_DOMAINS_ARE_CONFIGURED

1.35.10.43 2.4.2.2. HandshakeErrors

DOMAIN_ALIAS_DUPLICATION

Explanation: This error indicates that the domain alias was previously used to connect to a domain with a different domain id. This is a known situation when an existing participant is trying to connect to a freshly re-initialised domain.
Resolution: Carefully verify the connection settings.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: DOMAIN_ALIAS_DUPLICATION

DOMAIN_CRYPTO_HANDSHAKE_FAILED

Explanation: This error indicates that the domain is using crypto settings which are either not supported or not enabled on this participant.
Resolution: Consult the error message and adjust the supported crypto schemes of this participant.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: DOMAIN_CRYPTO_HANDSHAKE_FAILED

DOMAIN_HANDSHAKE_FAILED

Explanation: This error indicates that the participant to domain handshake has failed.
Resolution: Inspect the provided reason for more details and contact support.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: DOMAIN_HANDSHAKE_FAILED

DOMAIN_ID_MISMATCH

Explanation: This error indicates that the domain-id does not match the one that the participant expects. If this error happens on a first connect, then the domain id defined in the domain connection settings does not match the remote domain. If this happens on a reconnect, then the remote domain has been reset for some reason.
Resolution: Carefully verify the connection settings.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: DOMAIN_ID_MISMATCH

SERVICE_AGREEMENT_ACCEPTANCE_FAILED

Explanation: This error indicates that the domain requires the participant to accept a service agreement before connecting to it.
Resolution: Use the commands $participant.domains.get_agreement and $participant.domains.accept_agreement to accept the agreement.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: SERVICE_AGREEMENT_ACCEPTANCE_FAILED

1.35.10.44 2.4.2.3. ConnectionErrors

DOMAIN_IS_NOT_AVAILABLE

Explanation: This error results if the GRPC connection to the domain service fails with GRPC status UNAVAILABLE.
Resolution: Check your connection settings and ensure that the domain can really be reached.
Category: TransientServerFailure
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status UNAVAILABLE including a detailed error message.
Scaladocs: DOMAIN_IS_NOT_AVAILABLE

FAILED_TO_CONNECT_TO_SEQUENCER

Explanation: This error indicates that the participant failed to connect to the sequencer.
Resolution: Inspect the provided reason.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: FAILED_TO_CONNECT_TO_SEQUENCER

FAILED_TO_CONNECT_TO_SEQUENCERS

Explanation: This error indicates that the participant failed to connect to the sequencers.
Resolution: Inspect the provided reason.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: FAILED_TO_CONNECT_TO_SEQUENCERS

GRPC_CONNECTION_FAILURE

Explanation: This error indicates that the participant failed to connect due to a general GRPC error.
Resolution: Inspect the provided reason and contact support.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: GRPC_CONNECTION_FAILURE

PARTICIPANT_IS_NOT_ACTIVE

Explanation: This error indicates that the connecting participant has either not yet been activated by the domain operator. If the participant was previously successfully connected to the domain, then this error indicates that the domain operator has deactivated the participant.
Resolution: Contact the domain operator and inquire the permissions your participant node has on the given domain.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: PARTICIPANT_IS_NOT_ACTIVE

1.35.10.45 2.4.3. TrafficControlError

TRAFFIC_CONTROL_DOMAIN_ID_NOT_FOUND

Explanation: This error indicates that no available domain with that id could be found, and therefore no traffic state could be retrieved.
Resolution: Ensure that the participant is connected to the domain with the provided id.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: TRAFFIC_CONTROL_DOMAIN_ID_NOT_FOUND

TRAFFIC_CONTROL_STATE_NOT_FOUND

Explanation: This error indicates that the participant does not have a traffic state.
Resolution: Ensure that the the participant is connected to a domain with traffic control enabled, and that it has received at least one event from the domain since its connection.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: TRAFFIC_CONTROL_STATE_NOT_FOUND

1.35.10.46 2.5. AdminWorkflowServices

CAN_NOT_AUTOMATICALLY_VET_ADMIN_WORKFLOW_PACKAGE

Explanation: This error indicates that the admin workflow package could not be vetted. The admin workflows is a set of packages that are pre-installed and can be used for administrative processes. The error can happen if the participant is initialised manually but is missing the appropriate signing keys or certificates in order to issue new topology transactions within the participants namespace. The admin workflows can not be used until the participant has vetted the package.
Resolution: This error can be fixed by ensuring that an appropriate vetting transaction is issued in the name of this participant and imported into this participant node. If the corresponding certificates have been added after the participant startup, then this error can be fixed by either restarting the participant node, issuing the vetting transaction manually or re-uploading the Dar (leaving the vetAllPackages argument as true)
Category: BackgroundProcessDegradationWarning
Conveyance: This error is logged with log-level WARN on the server side.
Scaladocs: CAN_NOT_AUTOMATICALLY_VET_ADMIN_WORKFLOW_PACKAGE

1.35.10.47 2.6. RepairServiceError

CONTRACT_PURGE_ERROR

Explanation: A contract cannot be purged due to an error.
Resolution: Retry after operator intervention.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: CONTRACT_PURGE_ERROR

INVALID_ACS_SNAPSHOT_TIMESTAMP

Explanation: The participant does not support serving an ACS snapshot at the requested timestamp, likely because some concurrent processing has not yet finished.
Resolution: Make sure that the specified timestamp has been obtained from the participant in some way. If so, retry after a bit (possibly repeatedly).
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: INVALID_ACS_SNAPSHOT_TIMESTAMP

INVALID_ARGUMENT_REPAIR

Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: INVALID_ARGUMENT_REPAIR

UNAVAILABLE_ACS_SNAPSHOT

Explanation: The participant does not support serving an ACS snapshot at the requested timestamp because its database has already been pruned, e.g., by the continuous background pruning process.
Resolution: The snapshot at the requested timestamp is no longer available. Pick a more recent timestamp if possible.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: UNAVAILABLE_ACS_SNAPSHOT

UNSUPPORTED_PROTOCOL_VERSION_PARTICIPANT

Explanation: The participant does not support the requested protocol version.
Resolution: Specify a protocol version that the participant supports or upgrade the participant to a release that supports the requested protocol version.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: UNSUPPORTED_PROTOCOL_VERSION_PARTICIPANT

1.35.10.48 2.7. IndexErrors

1.35.10.49 2.7.1. DatabaseErrors

INDEX_DB_INVALID_RESULT_SET

Explanation: This error occurs if the result set returned by a query against the Index database is invalid.
Resolution: Contact support.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: INDEX_DB_INVALID_RESULT_SET

INDEX_DB_SQL_NON_TRANSIENT_ERROR

Explanation: This error occurs if a non-transient error arises when executing a query against the index database.
Resolution: Contact the participant operator.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: INDEX_DB_SQL_NON_TRANSIENT_ERROR

INDEX_DB_SQL_TRANSIENT_ERROR

Explanation: This error occurs if a transient error arises when executing a query against the index database.
Resolution: Re-submit the request.
Category: TransientServerFailure
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status UNAVAILABLE including a detailed error message.
Scaladocs: INDEX_DB_SQL_TRANSIENT_ERROR

1.35.10.50 2.8. PruningServiceError

INTERNAL_PRUNING_ERROR

Explanation: Pruning has failed because of an internal server error.
Resolution: Identify the error in the server log.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: INTERNAL_PRUNING_ERROR

NON_CANTON_OFFSET

Explanation: The supplied offset has an unexpected lengths.
Resolution: Ensure the offset has originated from this participant and is 9 bytes in length.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: NON_CANTON_OFFSET

PRUNING_NOT_SUPPORTED_IN_COMMUNITY_EDITION

Explanation: Pruning is not supported in the Community Edition.
Resolution: Upgrade to the Enterprise Edition.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: PRUNING_NOT_SUPPORTED_IN_COMMUNITY_EDITION

SHUTDOWN_INTERRUPTED_PRUNING

Explanation: Pruning has been aborted because the participant is shutting down.
Resolution: After the participant is restarted, the participant ensures that it is in a consistent state. Therefore no intervention is necessary. After the restart, pruning can be invoked again as usual to prune the participant up to the desired offset.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: SHUTDOWN_INTERRUPTED_PRUNING

UNSAFE_TO_PRUNE

Explanation: Pruning is not possible at the specified offset at the current time.
Resolution: Specify a lower offset or retry pruning after a while. Generally, you can only prune older events. In particular, the events must be older than the sum of mediator reaction timeout and participant timeout for every domain. And, you can only prune events that are older than the deduplication time configured for this participant. Therefore, if you observe this error, you either just prune older events or you adjust the settings for this participant. The error details field *safe_offset* contains the highest offset that can currently be pruned, if any.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: UNSAFE_TO_PRUNE

1.35.10.51 2.9. CantonPackageServiceError

PACKAGE_OR_DAR_REMOVAL_ERROR

Explanation: Errors raised by the Package Service on package removal.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: PACKAGE_OR_DAR_REMOVAL_ERROR

1.35.10.52 2.10. ParticipantReplicationServiceError

PARTICIPANT_REPLICATION_INTERNAL_ERROR

Explanation: Internal error emitted upon internal participant replication errors
Resolution: Contact support
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: PARTICIPANT_REPLICATION_INTERNAL_ERROR

PARTICIPANT_REPLICATION_NOT_SUPPORTED_BY_STORAGE

Explanation: Error emitted if the supplied storage configuration does not support replication.
Resolution: Use a participant storage backend supporting replication.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: PARTICIPANT_REPLICATION_NOT_SUPPORTED_BY_STORAGE

1.35.10.53 2.11. CommonErrors

REQUEST_TIME_OUT

Explanation: This rejection is given when a request processing status is not known and a time-out is reached.
Resolution: Retry for transient problems. If non-transient contact the operator as the time-out limit might be too short.
Category: DeadlineExceededRequestStateUnknown
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status DEADLINE_EXCEEDED including a detailed error message.
Scaladocs: REQUEST_TIME_OUT

SERVER_IS_SHUTTING_DOWN

Explanation: This rejection is given when the participant server is shutting down.
Resolution: Contact the participant operator.
Category: TransientServerFailure
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status UNAVAILABLE including a detailed error message.
Scaladocs: SERVER_IS_SHUTTING_DOWN

SERVICE_INTERNAL_ERROR

Explanation: This error occurs if one of the services encountered an unexpected exception.
Resolution: Contact support.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: SERVICE_INTERNAL_ERROR

SERVICE_NOT_RUNNING

Explanation: This rejection is given when the requested service has already been closed.
Resolution: Retry re-submitting the request. If the error persists, contact the participant operator.
Category: TransientServerFailure
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status UNAVAILABLE including a detailed error message.
Scaladocs: SERVICE_NOT_RUNNING

UNSUPPORTED_OPERATION

Explanation: This error category is used to signal that an unimplemented code-path has been triggered by a client or participant operator request.
Resolution: This error is caused by a participant node misconfiguration or by an implementation bug. Resolution requires participant operator intervention.
Category: InternalUnsupportedOperation
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status UNIMPLEMENTED without any details for security reasons.
Scaladocs: UNSUPPORTED_OPERATION

1.35.10.54 3. EthereumErrors

1.35.10.55 3.1. ConfigurationErrors

AHEAD_OF_HEAD

Explanation: This warning is logged on startup if the sequencer is configured to only start reading from a block that wasn't mined yet by the blockchain (e.g. sequencer is supposed to start reading from block 500, but the latest block is only 100). This is likely due to a misconfiguration.
Resolution: This issue frequently occurs when the blockchain is reset but the sequencer database configuration is not updated or the sequencer database (which persists the last block that was read by the sequencer) is not reset. Validate these settings and ensure that the sequencer is still reading from the same blockchain.
Category: BackgroundProcessDegradationWarning
Conveyance: This error is logged with log-level WARN on the server side.
Scaladocs: AHEAD_OF_HEAD

ATTEMPT_TO_CHANGE_IMMUTABLE_VALUE

Explanation: The sequencer smart contract has detected that a value that is immutable after being set for the first time (either the signing tolerance or the topology manager ID) was attempted to be changed. Most frequently this error occurs during testing when a Canton Ethereum sequencer process without persistence is restarted while pointing to the same smart sequencer contract. An Ethereum sequencer attempts to set the topology manager ID during initialization, however, without persistence the topology manager ID is randomly regenerated on the restart which leads to the sequencer attempting to change the topology manager ID in the sequencer smart contract.
Resolution: Deploy a new instance of the sequencer contract and configure the Ethereum sequencer to use that instance. If the errors occur because an Ethereum sequencer process is restarted without persistence, deploy a fresh instance of the sequencer contract and configure persistence for restarts.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: ATTEMPT_TO_CHANGE_IMMUTABLE_VALUE

BESU_VERSION_MISMATCH

Explanation: This error is logged when the sequencer detects that the version of the Besu client it connects to is not what is expected / supported.
Resolution: Either deploy the documented required version or set canton.parameters.non-standard-config = true.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: BESU_VERSION_MISMATCH

ETHEREUM_CANT_QUERY_VERSION

Explanation: As one of the first steps when initializing a Besu sequencer, Canton attempts to query the version (attribute) of the Sequencer.sol contract.
Resolution: Usually, the root cause of this is a deployment or configuration problem. Ensure that a Sequencer.sol contract is deployed on the configured address on the latest block when attempting to initialize the Canton Besu sequencer node. If this error persists, a malicious user may be attempting to interfere with the Ethereum network.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level WARN on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: ETHEREUM_CANT_QUERY_VERSION

MANY_BLOCKS_BEHIND_HEAD

Explanation: This error is logged when the sequencer is currently processing blocks that are very far behind the head of the blockchain of the connected Ethereum network. The Ethereum sequencer won't observe new transactions in the blockchain until it has caught up to the head. This may take a long time depending on the blockchain length and number of Canton transaction in the blocks. Empirically, we have observed that the Canton sequencer processes roughly 500 empty blocks/second. This may vary strongly for non-empty blocks. The sequencer logs once it has caught up to within *blocksBehindBlockchainHead* blocks behind the blockchain head.
Resolution: Wait until the sequencer has caught up to the head of the blockchain. Alternatively, consider changing the configuration of *block-to-read-from* of the Ethereum sequencer when initializing it against an Ethereum network that already mined a lot of blocks.
Category: BackgroundProcessDegradationWarning
Conveyance: This error is logged with log-level WARN on the server side.
Scaladocs: MANY_BLOCKS_BEHIND_HEAD

NOT_FREE_GAS_NETWORK

Explanation: This error is logged when during setup the sequencer detects that it isn't connected to a free-gas network. This usually leads to transactions silently being dropped by Ethereum nodes. You should only use a non-free-gas network, if you have configured an Ethereum wallet for the sequencer to use and have given it gas.
Resolution: Change the configuration of the Ethereum network to a free-gas network.
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: NOT_FREE_GAS_NETWORK

WRONG_EVM_BYTECODE

Explanation: Canton validates on startup that the configured address on the blockchain contains the EVM bytecode of the sequencer smart contract in the latest block. This error indicates that no bytecode or the wrong bytecode was found. This is a serious error and means that the sequencer can't sequence events.
Resolution: This frequently error occurs when updating the Canton system without updating the sequencer contract deployed on the blockchain. Validate that the sequencer contract corresponding to the current Canton release is deployed in the latest blockchain blocks on the configured address. Another common reason for this error is that the wrong contract address was configured.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level WARN on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: WRONG_EVM_BYTECODE

1.35.10.56 3.2. TransactionErrors

ETHEREUM_TRANSACTION_INVALID

Explanation: This error happens when the Sequencer Ethereum application reads a transaction from the blockchain which is malformed (e.g, invalid member, arguments aren't parseable or too large). This could happen if a malicious or faulty Ethereum Sequencer node is placing faulty data on the blockchain.
Resolution: Generally, Canton should recover automatically from this error. The faulty transactions are simply skipped by all non-malicious/non-faulty sequencers in a deterministic way, so the integrity of the event stream across sequencer nodes should be maintained. If you continue to see this error, investigate whether some of the sequencer nodes in the network are misbehaving.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: ETHEREUM_TRANSACTION_INVALID

ETHEREUM_TRANSACTION_RECEIPT_FETCHING_FAILED

Explanation: This error occurs when the Ethereum sequencer attempts to fetch the transaction receipt for a previously submitted transaction but receives an exception. Usually, this is caused by network errors, the Ethereum client node being overloaded or the Ethereum sequencer reaching its *transactionReceiptPollingAttempts* for a given transaction. The fetching of transaction receipts of submitted transactions is separate from the Ethereum sequencer's read-stream used to ingest new transactions. Thus, in this sense, this error is purely informative and can be caused by transient issues (such as a transient network outage). Note, that the Canton nonce manager refreshes his cache whenever this error occurs which may unblock stuck transactions with a too-high nonce.
Resolution: Usually, this error should resolve by itself. If you frequently see this error, ensure that the Ethereum account of the Ethereum sequencer is used by no one else and that the Ethereum client doesn't drop submitted transactions through being overloaded or reaching a full txpool. If this errors keeps occurring, please contact support.

Category: BackgroundProcessDegradationWarning
Conveyance: This error is logged with log-level WARN on the server side.
Scaladocs: ETHEREUM_TRANSACTION_RECEIPT_FETCHING_FAILED

ETHEREUM_TRANSACTION_SUBMISSION_FAILED

Explanation: This error is logged when the Sequencer Ethereum application receives an error when attempting to submit a transaction to the transaction pool of the Ethereum client. Common causes for this are network errors, or when the Ethereum account of the Sequencer Ethereum application is used by another application. Less commonly, this error might also indirectly be caused if the transaction pool of the Ethereum client is full or flushed.
Resolution: Generally, Canton should recover automatically from this error. If you continue to see this error, investigate possible root causes such as poor network connections, if the Ethereum wallet of the Ethereum Sequencer application is being reused by another application, and the health of the Ethereum client.
Category: BackgroundProcessDegradationWarning
Conveyance: This error is logged with log-level WARN on the server side.
Scaladocs: ETHEREUM_TRANSACTION_SUBMISSION_FAILED

1.35.10.57 4. TopologyManagementErrorGroup

1.35.10.58 4.1. TopologyManagerError

DUPLICATE_TOPOLOGY_TRANSACTION

Explanation: This error indicates that a transaction has already been added previously.
Resolution: Nothing to do as the transaction is already registered. Note however that a revocation is + final. If you want to re-enable a statement, you need to re-issue an new transaction.
Category: InvalidGivenCurrentSystemStateResourceExists
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ALREADY_EXISTS including a detailed error message.
Scaladocs: DUPLICATE_TOPOLOGY_TRANSACTION

INCREASE_OF_LEDGER_TIME_RECORD_TIME_TOLERANCE

Explanation: This error indicates that it has been attempted to increase the `ledgerTimeRecordTimeTolerance` domain parameter in an insecure manner. Increasing this parameter may disable security checks and can therefore be a security risk.
Resolution: Make sure that the new value of `ledgerTimeRecordTimeTolerance` is at most half of the `mediatorDeduplicationTimeout` domain parameter. Use `myDomain.service.set_ledger_time_record_time_tolerance` for securely increasing ledgerTimeRecordTimeTolerance. Alternatively, add the `force = true` flag to your command, if security is not a concern for you. The security checks will be effective again after twice the new value of `ledgerTimeRecordTimeTolerance`. Using `force = true` is safe upon domain bootstrapping.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: INCREASE_OF_LEDGER_TIME_RECORD_TIME_TOLERANCE

INVALID_DOMAIN

Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: INVALID_DOMAIN

INVALID_TOPOLOGY_TX_SIGNATURE_ERROR

Explanation: This error indicates that the uploaded signed transaction contained an invalid signature.
Resolution: Ensure that the transaction is valid and uses a crypto version understood by this participant.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: INVALID_TOPOLOGY_TX_SIGNATURE_ERROR

NO_APPROPRIATE_SIGNING_KEY_IN_STORE

Explanation: This error results if the topology manager did not find a secret key in its store to authorize a certain topology transaction.
Resolution: Inspect your topology transaction and your secret key store and check that you have the appropriate certificates and keys to issue the desired topology transaction. If the list of candidates is empty, then you are missing the certificates.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: NO_APPROPRIATE_SIGNING_KEY_IN_STORE

NO_CORRESPONDING_ACTIVE_TX_TO_REVOKE

Explanation: This error indicates that the attempt to add a removal transaction was rejected, as the mapping / element affecting the removal did not exist.
Resolution: Inspect the topology state and ensure the mapping and the element id of the active transaction you are trying to revoke matches your revocation arguments.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: NO_CORRESPONDING_ACTIVE_TX_TO_REVOKE

PUBLIC_KEY_NOT_IN_STORE

Explanation: This error indicates that a command contained a fingerprint referring to a public key not being present in the public key store.
Resolution: Upload the public key to the public key store using $node.keys.public.load(.) before retrying.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: PUBLIC_KEY_NOT_IN_STORE

REMOVING_KEY_DANGLING_TRANSACTIONS_MUST_BE_FORCED

Explanation: This error indicates that the attempted key removal would create dangling topology transactions, making the node unusable.
Resolution: Add the *force = true* flag to your command if you are really sure what you are doing.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: REMOVING_KEY_DANGLING_TRANSACTIONS_MUST_BE_FORCED

REMOVING_LAST_KEY_MUST_BE_FORCED

Explanation: This error indicates that the attempted key removal would remove the last valid key of the given entity, making the node unusable.
Resolution: Add the *force = true* flag to your command if you are really sure what you are doing.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: REMOVING_LAST_KEY_MUST_BE_FORCED

SECRET_KEY_NOT_IN_STORE

Explanation: This error indicates that the secret key with the respective fingerprint can not be found.
Resolution: Ensure you only use fingerprints of secret keys stored in your secret key store.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: SECRET_KEY_NOT_IN_STORE

SERIAL_MISMATCH

Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: SERIAL_MISMATCH

TOPOLOGY_MANAGER_ALARM

Explanation: The topology manager has received a malformed message from another node.
Resolution: Inspect the error message for details.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: TOPOLOGY_MANAGER_ALARM

TOPOLOGY_MANAGER_INTERNAL_ERROR

Explanation: This error indicates that there was an internal error within the topology manager.
Resolution: Inspect error message for details.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: TOPOLOGY_MANAGER_INTERNAL_ERROR

TOPOLOGY_MAPPING_ALREADY_EXISTS

Explanation: This error indicates that a topology transaction would create a state that already exists and has been authorized with the same key.
Resolution: Your intended change is already in effect.
Category: InvalidGivenCurrentSystemStateResourceExists
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ALREADY_EXISTS including a detailed error message.
Scaladocs: TOPOLOGY_MAPPING_ALREADY_EXISTS

UNAUTHORIZED_TOPOLOGY_TRANSACTION

Explanation: This error indicates that the attempt to add a transaction was rejected, as the signing key is not authorized within the current state.
Resolution: Inspect the topology state and ensure that valid namespace or identifier delegations of the signing key exist or upload them before adding this transaction.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: UNAUTHORIZED_TOPOLOGY_TRANSACTION

1.35.10.59 4.1.1. DomainTopologyManagerError

ALIEN_DOMAIN_ENTITIES

Explanation: This error is returned if a transaction attempts to add keys for alien domain manager or sequencer entities to this domain topology manager.
Resolution: Use a participant topology manager if you want to manage foreign domain keys
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: ALIEN_DOMAIN_ENTITIES

FAILED_TO_ADD_MEMBER

Explanation: This error indicates an external issue with the member addition hook.
Resolution: Consult the error details.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: FAILED_TO_ADD_MEMBER

MALICOUS_OR_FAULTY_ONBOARDING_REQUEST

Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: MALICOUS_OR_FAULTY_ONBOARDING_REQUEST

PARTICIPANT_NOT_INITIALIZED

Explanation: This error is returned if a domain topology manager attempts to activate a participant without having all necessary data, such as keys or domain trust certificates.
Resolution: Register the necessary keys or trust certificates and try again.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: PARTICIPANT_NOT_INITIALIZED

WRONG_DOMAIN

Explanation: This error is returned if a transaction restricted to a domain should be added to another domain.
Resolution: Recreate the content of the transaction with a correct domain identifier.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: WRONG_DOMAIN

WRONG_PROTOCOL_VERSION

Explanation: This error is returned if a transaction has a protocol version different than the one spoken on the domain.
Resolution: Recreate the transaction with a correct protocol version.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: WRONG_PROTOCOL_VERSION

1.35.10.60 4.1.2. ParticipantTopologyManagerError

CANNOT_VET_DUE_TO_MISSING_PACKAGES

Explanation: This error indicates that a package vetting command failed due to packages not existing locally. This can be due to either the packages not being present or their dependencies being missing. When vetting a package, the package must exist on the participant, as otherwise the participant will not be able to process a transaction relying on a particular package.
Resolution: Ensure that the package exists locally before issuing such a transaction.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: CANNOT_VET_DUE_TO_MISSING_PACKAGES

DANGEROUS_KEY_USE_COMMAND_REQUIRES_FORCE

Explanation: This error indicates that a dangerous owner to key mapping authorization was rejected. This is the case if a command is run that could break a participant. If the command was run to assign a key for the given participant, then the command was rejected because the key is not in the participants private store. If the command is run on a participant to issue transactions for another participant, then such commands must be run with force, as they are very dangerous and could easily break the participant. As an example, if we assign an encryption key to a participant that the participant does not have, then the participant will be unable to process an incoming transaction. Therefore we must be very careful to not create such situations.
Resolution: Set force=true if you really know what you are doing.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: DANGEROUS_KEY_USE_COMMAND_REQUIRES_FORCE

DANGEROUS_VETTING_COMMANDS_REQUIRE_FORCE

Explanation: This error indicates that a dangerous package vetting command was rejected. This is the case if a vetting command, if not run correctly, could potentially lead to a ledger fork. The vetting authorization checks the participant for the presence of the given set of packages (including their dependencies) and allows only to vet for the given participant id. In rare cases where a more centralised topology manager is used, this behaviour can be overridden with force. However, if a package is vetted but not present on the participant, the participant will refuse to process any transaction of the given domain until the problematic package has been uploaded.
Resolution: Set force=true if you really know what you are doing.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: DANGEROUS_VETTING_COMMANDS_REQUIRE_FORCE

DEPENDENCIES_NOT_VETTED

Explanation: This error indicates a vetting request failed due to dependencies not being vetted. On every vetting request, the set supplied packages is analysed for dependencies. The system requires that not only the main packages are vetted explicitly but also all dependencies. This is necessary as not all participants are required to have the same packages installed and therefore not every participant can resolve the dependencies implicitly.
Resolution: Vet the dependencies first and then repeat your attempt.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: DEPENDENCIES_NOT_VETTED

DISABLE_PARTY_WITH_ACTIVE_CONTRACTS_REQUIRES_FORCE

Explanation: This error indicates that a dangerous PartyToParticipant mapping deletion was rejected. If the command is run and there are active contracts where the party is a stakeholder these contracts will become inoperable and will never get pruned, leaking storage.
Resolution: Set force=true if you really know what you are doing.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: DISABLE_PARTY_WITH_ACTIVE_CONTRACTS_REQUIRES_FORCE

UNINITIALIZED_PARTICIPANT

Explanation: This error indicates that a request involving topology management was attempted on a participant that is not yet initialised. During initialisation, only namespace and identifier delegations can be managed.
Resolution: Initialise the participant and retry.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: UNINITIALIZED_PARTICIPANT

1.35.10.61 4.1.3. Domain

DOMAIN_NODE_INITIALISATION_FAILED

Explanation: This error indicates that the initialisation of a domain node failed due to invalid arguments.
Resolution: Consult the error details.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: DOMAIN_NODE_INITIALISATION_FAILED

1.35.10.62 4.1.3.1. GrpcSequencerAuthenticationService

CLIENT_AUTHENTICATION_FAULTY

Explanation: This error indicates that a client failed to authenticate with the sequencer due to a reason possibly pointing out to faulty or malicious behaviour. The message is logged on the server in order to support an operator to provide explanations to clients struggling to connect.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side.
Scaladocs: CLIENT_AUTHENTICATION_FAULTY

CLIENT_AUTHENTICATION_REJECTED

Explanation: This error indicates that a client failed to authenticate with the sequencer. The message is logged on the server in order to support an operator to provide explanations to clients struggling to connect.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side.
Scaladocs: CLIENT_AUTHENTICATION_REJECTED

1.35.10.63 4.2. DomainTopologySender

TOPOLOGY_DISPATCHING_DEGRADATION

Explanation: This warning occurs when the topology dispatcher experiences timeouts while trying to register topology transactions.
Resolution: This error should normally self-recover due to retries. If issue persist, contact support and investigate system state.
Category: BackgroundProcessDegradationWarning
Conveyance: This error is logged with log-level WARN on the server side.
Scaladocs: TOPOLOGY_DISPATCHING_DEGRADATION

TOPOLOGY_DISPATCHING_INTERNAL_ERROR

Explanation: This error is emitted if there is a fundamental, un-expected situation during topology dispatching. In such a situation, the topology state of a newly onboarded participant or of all domain members might become outdated
Resolution: Contact support.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: TOPOLOGY_DISPATCHING_INTERNAL_ERROR

1.35.10.64 5. ConfigErrors

CANNOT_PARSE_CONFIG_FILES

Explanation: This error is usually thrown because a config file doesn't contain configs in valid HOCON format. The most common cause of an invalid HOCON format is a forgotten bracket.
Resolution: Make sure that all files are in valid HOCON format.
Category: InvalidIndependentOfSystemState
Conveyance: Config errors are logged and output to stdout if starting Canton with a given configuration fails
Scaladocs: CANNOT_PARSE_CONFIG_FILES

CANNOT_READ_CONFIG_FILES

Explanation: This error is usually thrown when Canton can't find a given configuration file.
Resolution: Make sure that the path and name of all configuration files is correctly specified.
Category: InvalidIndependentOfSystemState
Conveyance: Config errors are logged and output to stdout if starting Canton with a given configuration fails
Scaladocs: CANNOT_READ_CONFIG_FILES

CONFIG_SUBSTITUTION_ERROR

Resolution: A common cause of this error is attempting to use an environment variable that isn't defined within a config-file.
Category: InvalidIndependentOfSystemState
Conveyance: Config errors are logged and output to stdout if starting Canton with a given configuration fails
Scaladocs: CONFIG_SUBSTITUTION_ERROR

CONFIG_VALIDATION_ERROR

Category: InvalidIndependentOfSystemState
Conveyance: Config errors are logged and output to stdout if starting Canton with a given configuration fails
Scaladocs: CONFIG_VALIDATION_ERROR

GENERIC_CONFIG_ERROR

Resolution: In general, this can be one of many errors since this is the 'miscellaneous category' of configuration errors. One of the more common errors in this category is an 'unknown key' error. This error usually means that a keyword that is not valid (e.g. it may have a typo 'bort' instead of 'port'), or that a valid keyword at the wrong part of the configuration hierarchy was used (e.g. to enable database replication for a participant, the correct configuration is *canton.participants.participant2.replication.enabled = true* and not *canton.participants.replication.enabled = true*). Please refer to the scaladoc of either *CantonEnterpriseConfig* or *CantonCommunityConfig* (depending on whether the community or enterprise version is used) to find the valid configuration keywords and the correct position in the configuration hierarchy.
Category: InvalidIndependentOfSystemState
Conveyance: Config errors are logged and output to stdout if starting Canton with a given configuration fails
Scaladocs: GENERIC_CONFIG_ERROR

NO_CONFIG_FILES

Category: InvalidIndependentOfSystemState
Conveyance: Config errors are logged and output to stdout if starting Canton with a given configuration fails
Scaladocs: NO_CONFIG_FILES

1.35.10.65 6. CommandErrors

CONSOLE_COMMAND_INTERNAL_ERROR

Category: SystemInternalAssumptionViolated
Conveyance: These errors are shown as errors on the console.
Scaladocs: CONSOLE_COMMAND_INTERNAL_ERROR

CONSOLE_COMMAND_TIMED_OUT

Category: SystemInternalAssumptionViolated
Conveyance: These errors are shown as errors on the console.
Scaladocs: CONSOLE_COMMAND_TIMED_OUT

NODE_NOT_STARTED

Category: InvalidGivenCurrentSystemStateOther
Conveyance: These errors are shown as errors on the console.
Scaladocs: NODE_NOT_STARTED

1.35.10.66 7. DatabaseStorageError

DB_CONNECTION_LOST

Explanation: This error indicates that the connection to the database has been lost.
Resolution: Inspect error message for details.
Category: BackgroundProcessDegradationWarning
Conveyance: This error is logged with log-level WARN on the server side.
Scaladocs: DB_CONNECTION_LOST

DB_STORAGE_DEGRADATION

Explanation: This error indicates that degradation of database storage components.
Resolution: This error indicates performance degradation. The error occurs when a database task has been rejected, typically due to having a too small task queue. The task will be retried after a delay. If this error occurs frequently, however, you may want to consider increasing the task queue. (Config parameter: canton.<path-to-my-node>.storage.config.queueSize).
Category: BackgroundProcessDegradationWarning
Conveyance: This error is logged with log-level WARN on the server side.

Scaladocs: DB_STORAGE_DEGRADATION

1.35.10.67 8. HandshakeErrors

DEPRECATED_PROTOCOL_VERSION

Explanation: This error is logged or returned if a participant or domain are using deprecated protocol versions. Deprecated protocol versions might not be secure anymore.
Resolution: Migrate to a new domain that uses the most recent protocol version.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: DEPRECATED_PROTOCOL_VERSION

1.35.10.68 9. FabricErrors

1.35.10.69 9.1. ConfigurationErrors

FABRIC_AHEAD_OF_HEAD

Explanation: This warning is logged on startup if the sequencer is configured to only start reading from a block that wasn't ordered yet by the blockchain (e.g. sequencer is supposed to start reading from block 500, but the latest block is only 100). This is likely due to a misconfiguration.
Resolution: This issue frequently occurs when the blockchain is reset but the sequencer database configuration is not updated or the sequencer database (which persists the last block that was read by the sequencer) is not reset. Validate these settings and ensure that the sequencer is still reading from the same blockchain.
Category: BackgroundProcessDegradationWarning
Conveyance: This error is logged with log-level WARN on the server side.
Scaladocs: FABRIC_AHEAD_OF_HEAD

FABRIC_MANY_BLOCKS_BEHIND_HEAD

Explanation: This error is logged when the sequencer is currently processing blocks that are very far behind the head of the blockchain of the connected Fabric network. The Fabric sequencer won't observe new transactions in the blockchain until it has caught up to the head. This may take a long time depending on the blockchain length and number of Canton transaction in the blocks. Empirically, we have observed that the Canton sequencer processes roughly 500 empty blocks/second. This may vary strongly for non-empty blocks.
Resolution: Change the configuration of *startBlockHeight* for the Fabric sequencer when working with an existing (not fresh) Fabric network. Alternatively, wait until the sequencer has caught up to the head of the blockchain.
Category: BackgroundProcessDegradationWarning
Conveyance: This error is logged with log-level WARN on the server side.
Scaladocs: FABRIC_MANY_BLOCKS_BEHIND_HEAD

1.35.10.70 9.2. TransactionErrors

FABRIC_TRANSACTION_INVALID

Explanation: This error happens when the Sequencer Fabric application reads a transaction from the blockchain which is malformed (e.g, missing arguments, arguments aren't parseable or too large). This could happen if a malicious or faulty Fabric Sequencer node is placing faulty data on the blockchain.
Resolution: Generally, Canton should recover automatically from this error. The faulty transactions are simply skipped by all non-malicious/non-faulty sequencers in a deterministic way, so the integrity of the event stream across sequencer nodes should be maintained. If you continue to see this error, investigate whether some of the sequencer nodes in the network are misbehaving.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: FABRIC_TRANSACTION_INVALID

FABRIC_TRANSACTION_PROPOSAL_SUBMISSION_FAILED

Explanation: An error happened with the Fabric transaction proposal submissions possibly due to some of the peers being down or due to network issues. Thus won't stop the transaction workflow, because there might still be enough successful responses to satisfy the endorsement policy. Therefore the transaction might still go through successfully despite this being logged.
Resolution: Generally, Canton should recover automatically from this error. If you continue to see this error, investigate possible root causes such as poor network connections, if the Fabric sequencer is properly configured with enough peers and if they are running.
Category: TransientServerFailure
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status UNAVAILABLE including a detailed error message.
Scaladocs: FABRIC_TRANSACTION_PROPOSAL_SUBMISSION_FAILED

FABRIC_TRANSACTION_SUBMISSION_FAILED

Explanation: This error is logged when the Sequencer Fabric application receives an error during any of the transaction flow steps that prevents the submission of a transaction over the Fabric client. Common causes for this are network errors, peers that are down or that there aren't enough configured endorsers.
Resolution: Generally, Canton should recover automatically from this error. If you continue to see this error, investigate possible root causes such as poor network connections, if the Fabric sequencer is properly configured with enough peers and if they are running.
Category: TransientServerFailure
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status UNAVAILABLE including a detailed error message.
Scaladocs: FABRIC_TRANSACTION_SUBMISSION_FAILED

1.35.10.71 10. SequencerError

INVALID_ACKNOWLEDGEMENT_SIGNATURE

Explanation: This error indicates that the sequencer has detected an invalid acknowledgement request signature. This most likely indicates that the request is bogus and has been created by a malicious sequencer. So it will not get processed.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: INVALID_ACKNOWLEDGEMENT_SIGNATURE

INVALID_ACKNOWLEDGEMENT_TIMESTAMP

Explanation: This error indicates that the member has acknowledged a timestamp that is after the events it has received. This violates the sequencing protocol.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: INVALID_ACKNOWLEDGEMENT_TIMESTAMP

INVALID_ENVELOPE_SIGNATURE

Explanation: This error indicates that the sequencer has detected an invalid envelope signature in the submission request. This most likely indicates that the request is bogus and has been created by a malicious sequencer. So it will not get processed.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: INVALID_ENVELOPE_SIGNATURE

INVALID_LEDGER_EVENT

Explanation: The sequencer has detected that some event that was placed on the ledger cannot be parsed. This may be due to some sequencer node acting maliciously or faulty. The event is ignored and processing continues as usual.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: INVALID_LEDGER_EVENT

INVALID_SEQUENCER_PRUNING_REQUEST_ON_CHAIN

Explanation: This error indicates that some sequencer node has distributed an invalid sequencer pruning request via the blockchain. Either the sequencer nodes got out of sync or one of the sequencer nodes is buggy. The sequencer node will stop processing to prevent the danger of severe data corruption.
Resolution: Stop using the domain involving the sequencer nodes. Contact support.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: INVALID_SEQUENCER_PRUNING_REQUEST_ON_CHAIN

INVALID_SUBMISSION_REQUEST_SIGNATURE

Explanation: This error indicates that the sequencer has detected an invalid submission request signature. This most likely indicates that the request is bogus and has been created by a malicious sequencer. So it will not get processed.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: INVALID_SUBMISSION_REQUEST_SIGNATURE

MAX_REQUEST_SIZE_EXCEEDED

Explanation: This error means that the request size has exceeded the configured value maxRequestSize.
Resolution: Send smaller requests or increase the maxRequestSize in the domain parameters
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: MAX_REQUEST_SIZE_EXCEEDED

MISSING_SUBMISSION_REQUEST_SIGNATURE_TIMESTAMP

Explanation: This error indicates that the sequencer has detected that the signed submission request being processed is missing a signature timestamp. It indicates that the sequencer node that placed the request is not following the protocol as there should always be a defined timestamp. This request will not get processed.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: MISSING_SUBMISSION_REQUEST_SIGNATURE_TIMESTAMP

MULTIPLE_MEDIATOR_RECIPIENTS

Explanation: This error indicates that the participant is trying to send envelopes to multiple mediators or mediator groups in the same submission request. This most likely indicates that the request is bogus and has been created by a malicious sequencer. So it will not get processed.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: MULTIPLE_MEDIATOR_RECIPIENTS

1.35.10.72 11. EnterpriseGrpcVaultServiceError

INVALID_KMS_KEY_ID

Explanation: Selected KMS key id is invalid
Resolution: Specify a valid key id and retry.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: INVALID_KMS_KEY_ID

NO_ENCRYPTED_PRIVATE_KEY_STORE_ERROR

Explanation: Node is not running an encrypted private store
Resolution: Verify that an encrypted store and KMS config are set for this node.
Category: InternalUnsupportedOperation
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status UNIMPLEMENTED without any details for security reasons.
Scaladocs: NO_ENCRYPTED_PRIVATE_KEY_STORE_ERROR

REGISTER_KMS_KEY_INTERNAL_ERROR

Explanation: Internal error emitted upon failing to register a KMS key in Canton
Resolution: Contact support
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: REGISTER_KMS_KEY_INTERNAL_ERROR

WRAPPER_KEY_ALREADY_IN_USE_ERROR

Explanation: Selected wrapper key id for rotation is already in use
Resolution: Select a different key id and retry.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: WRAPPER_KEY_ALREADY_IN_USE_ERROR

WRAPPER_KEY_DISABLED_OR_DELETED_ERROR

Explanation: Selected wrapper key id for rotation cannot be used because key is disabled or set to be deleted
Resolution: Specify a key id from an active key and retry.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: WRAPPER_KEY_DISABLED_OR_DELETED_ERROR

WRAPPER_KEY_NOT_EXIST_ERROR

Explanation: Selected wrapper key id for rotation does not match any existing KMS key
Resolution: Specify a key id that matches an existing KMS key and retry.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status NOT_FOUND including a detailed error message.
Scaladocs: WRAPPER_KEY_NOT_EXIST_ERROR

WRAPPER_KEY_ROTATION_INTERNAL_ERROR

Explanation: Internal error emitted upon internal wrapper key rotation errors
Resolution: Contact support
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: WRAPPER_KEY_ROTATION_INTERNAL_ERROR

1.35.10.73 12. MediatorError

MEDIATOR_INTERNAL_ERROR

Explanation: Request processing failed due to a violation of internal invariants. It indicates a bug at the mediator.
Resolution: Contact support.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.

Scaladocs: MEDIATOR_INTERNAL_ERROR

MEDIATOR_INVALID_MESSAGE

Explanation: The mediator has received an invalid message (request or response). The message will be discarded. As a consequence, the underlying request may be rejected. No corruption of the ledger is to be expected.
Resolution: Address the cause of the error. Let the submitter retry the command.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level WARN on the server side and exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message.
Scaladocs: MEDIATOR_INVALID_MESSAGE

MEDIATOR_RECEIVED_MALFORMED_MESSAGE

Explanation: The mediator has received a malformed message. This may occur due to a bug at the sender of the message. The message will be discarded. As a consequence, the underlying request may be rejected. No corruption of the ledger is to be expected.
Resolution: Contact support.
Category: SecurityAlert
Conveyance: This error is logged with log-level WARN on the server side. It is exposed on the API with grpc-status INVALID_ARGUMENT without any details for security reasons.
Scaladocs: MEDIATOR_RECEIVED_MALFORMED_MESSAGE

MEDIATOR_SAYS_TX_TIMED_OUT

Explanation: This rejection indicates that the transaction has been rejected by the mediator as it didn't receive enough confirmations within the participant response timeout.
Resolution: Check that all involved participants are available and not overloaded.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status ABORTED including a detailed error message.
Scaladocs: MEDIATOR_SAYS_TX_TIMED_OUT

1.35.10.74 13. ProtoDeserializationError

PROTO_DESERIALIZATION_FAILURE

Explanation: This error indicates that an incoming administrative command could not be processed due to a malformed message.
Resolution: Inspect the error details and correct your application
Category: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level INFO on the server side and exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message.
Scaladocs: PROTO_DESERIALIZATION_FAILURE

1.35.10.75 14. ResilientSequencerSubscription

SEQUENCER_FORK_DETECTED

Explanation: This error is logged when a sequencer client determined a ledger fork, where a sequencer node responded with different events for the same timestamp / counter. Whenever a client reconnects to a domain, it will start with the last message received and compare whether that last message matches the one it received previously. If not, it will report with this error. A ledger fork should not happen in normal operation. It can happen if the backups have been taken in a wrong order and e.g. the participant was more advanced than the sequencer.
Resolution: You can recover by restoring the system with a correctly ordered backup. Please consult the respective sections in the manual.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. It is exposed on the API with grpc-status INTERNAL without any details for security reasons.
Scaladocs: SEQUENCER_FORK_DETECTED

SEQUENCER_SUBSCRIPTION_LOST

Explanation: This warning is logged when a sequencer subscription is interrupted. The system will keep on retrying to reconnect indefinitely.
Resolution: Monitor the situation and contact the server operator if the issues does not resolve itself automatically.
Category: BackgroundProcessDegradationWarning
Conveyance: This error is logged with log-level WARN on the server side.
Scaladocs: SEQUENCER_SUBSCRIPTION_LOST

1.35.10.76 15. Clock

SYSTEM_CLOCK_RUNNING_BACKWARDS

Explanation: This error is emitted if the unique time generation detects that the host system clock is lagging behind the unique time source by more than a second. This can occur if the system processes more than 2e6 events per second (unlikely) or when the underlying host system clock is running backwards.
Resolution: Inspect your host system. Generally, the unique time source is not negatively affected by a clock moving backwards and will keep functioning. Therefore, this message is just a warning about something strange being detected.
Category: BackgroundProcessDegradationWarning
Conveyance: This error is logged with log-level WARN on the server side.
Scaladocs: SYSTEM_CLOCK_RUNNING_BACKWARDS

1.36 Troubleshooting

1.36.1 Error: "<X> is not authorized to commit an update"

This error occurs when there are multiple obligables on a contract.

A cornerstone of Daml is that you cannot create a contract that will force some other party (or parties) into an obligation. This error means that a party is trying to do something that would force another parties into an agreement without their consent.

To solve this, make sure each party is entering into the contract freely by exercising a choice. A good way of ensuring this is the initial and accept pattern: see the Daml patterns for more details.

1.36.2 Error: "Argument is not of serializable type"

This error occurs when you're using a function as a parameter to a template. For example, here is a contract that creates a `Payout` controller by a receiver's supervisor:

```
template SupervisedPayout
  with
    supervisor : Party -> Party
    receiver   : Party
    giver      : Party
    amount     : Decimal
  where
    signatory giver
    observer (supervisor receiver)
    choice SupervisedPayout_Call
      : ContractId Payout
      controller supervisor receiver
      do create Payout with giver; receiver; amount
```

Hovering over the compilation error displays:

```
[Type checker] Argument expands to non-serializable type Party -> Party.
```

1.36.3 Modeling Questions

1.36.3.1 How To Model an Agreement With Another Party

To enter into an agreement, create a contract from a template that has explicit `signatory` and `agreement` statements.

You'll need to use a series of contracts that give each party the chance to consent, via a contract choice.

Because of the rules that Daml enforces, it is not possible for a single party to create an instance of a multi-party agreement. This is because such a creation would force the other parties into that agreement, without giving them a choice to enter it or not.

1.36.3.2 How To Model Rights

Use a contract choice to model a right. A party exercises that right by exercising the choice.

1.36.3.3 How To Void a Contract

To allow voiding a contract, provide a choice that does not create any new contracts. Daml contracts are archived (but not deleted) when a consuming choice is made - so exercising the choice effectively voids the contract.

However, you should bear in mind who is allowed to void a contract, especially without the re-sought consent of the other signatories.

1.36.3.4 How To Represent Off-ledger Parties

You'd need to do this if you can't set up all parties as ledger participants, because the Daml `Party` type gets associated with a cryptographic key and can so only be used with parties that have been set up accordingly.

To model off-ledger parties in Daml, they must be represented on-ledger by a participant who can sign on their behalf. You could represent them with an ordinary `Text` argument.

This isn't very private, so you could use a numeric ID/an accountId to identify the off-ledger client.

1.36.3.5 How To Limit a Choice by Time

Some rights have a time limit: either a time by which it must be exercised or a time before which it cannot be exercised.

You can use `getTime` to get the current time, and compare your desired time to it. Use `assert` to abort the choice if your time condition is not met.

1.36.3.6 How To Model a Mandatory Action

If you want to ensure that a party takes some action within a given time period. Might want to incur a penalty if they don't - because that would breach the contract.

For example: an Invoice that must be paid by a certain date, with a penalty (could be something like an added interest charge or a penalty fee). To do this, you could have a time-limited Penalty choice that can only be exercised *after* the time period has expired.

However, note that the penalty action can only ever create another contract on the ledger, which represents an agreement between all parties that the initial contract has been breached. Ultimately, the recourse for any breach is legal action of some kind. What Daml provides is provable violation of the agreement.

1.36.3.7 When to Use Optional

The `Optional` type, from the standard library, to indicate that a value is optional, i.e, that in some cases it may be missing.

In functional languages, `Optional` is a better way of indicating a missing value than using the more familiar value NULL, present in imperative languages like Java.

To use `Optional`, include `Optional.daml` from the standard library:

```
import DA.Optional
```

Then, you can create `Optional` values like this:

```
Some "Some text"    -- Optional value exists.
```

```
None                -- Optional value does not exist.
```

You can test for existence in various ways:

```
-- isSome returns True if there is a value.
if isSome m
  then "Yes"
  else "No"
```

```
-- The inverse is isNone.
if isNone m
  then "No"
  else "Yes"
```

If you need to extract the value, use the `optional` function.

It returns a value of a defined type, and takes a `Optional` value and a function that can transform the value contained in a `Some` value of the `Optional` to that type. If it is missing `optional` also takes a value of the return type (the default value), which will be returned if the `Optional` value is `None`

```
let f = \ (i : Int) -> "The number is " <> (show i)
let t = optional "No number" f someValue
```

If `optionalValue` is `Some 5`, the value of `t` would be `"The number is 5"`. If it was `None`, `t` would be `"No number"`. Note that with `optional`, it is possible to return a different type from that contained in the `Optional` value. This makes the `Optional` type very flexible.

There are many other functions in Optional.daml that let you perform familiar functional operations on structures that contain `Optional` values - such as `map`, `filter`, etc. on `Lists` of `Optional` values.

1.36.4 Testing Questions

1.36.4.1 How To Test That a Contract Is Visible to a Party

Use `queryContractId`: its first argument is a party, and the second is a `ContractId`. If the contract corresponding to that `ContractId` exists and is visible to the party, the result will be wrapped in `Some`, otherwise the result will be `None`.

Use a `submit` block and a `fetch` operation. The `submit` block tests that the contract (as a `ContractId`) is visible to that party, and the `fetch` tests that it is valid, i.e., that the contract does exist.

For example, if we wanted to test for the existence and visibility of an `Invoice`, visible to 'Alice', whose ContractId is bound to *invoiceCid*, we could say:

```
Some result <- alice `queryContractId` invoiceCid
```

Note that we pattern match on the `Some` constructor. If the contract doesn't exist or is not visible to 'Alice', the test will fail with a pattern match error.

Now that the contract is bound to a variable, we can check whether it has some expected values:

```
result === Invoice with
  payee = alice
  payer = acme
  amount = 130.0
  service = "A job well done"
  timeLimit = datetime 1970 Feb 20 0 0 0
```

1.36.4.2 How To Test That an Update Action Cannot Be Committed

Use the `submitMustFail` function. This is similar in form to the `submit` function, but is an assertion that an update will fail if attempted by some Party.

1.37 Getting Help

Have questions or feedback? You're in the right place.

> **Questions: Forum**
> For how do I? , why does something work this way or I've got a programming problem I'm trying to solve questions, the `Questions` category on our forum is the best place to ask.
> If you're not sure what makes a good question, take a look at our guide on the topic.
> **Feedback: Forum**
> If you want to give feedback, you can make a topic in the `General` category on our forum.

When you're in the community Forum or on Stack Overflow, please keep to our Code of Conduct.

1.37.1 Support Expectations

For Daml Open Source users:

Timing: You can enjoy the support of the community, which is provided for you out of their own good will and free time. On top of that, a Digital Asset employee will try to reply to unanswered questions within two business days.
Business days are affected by public holidays. Engineers contributing to Daml are mostly located in Zurich and New York, so please be mindful of the public holidays in those locations (timeanddate.com maintains an unofficial list of holidays for both Switzerland and the United States).
Public support: We offer public support in the `Questions` category on our forum.
We can't answer questions in private messages or over email, so please only ask questions in public forums.
Level of support: We're happy to answer questions about error messages you're encountering, or discuss Daml design questions. However, we can't provide more extensive consultation on how to build your Daml application or the languages, frameworks, libraries and tools you may use to build it.

Digital Asset offers paid private support, which can be accessed here. Digital Asset further offers consultation on how to build your Daml application; please contact us for pricing.

1.38 Portability, Compatibility, and Support Durations

The Daml Ecosystem offers a number of forward and backward compatibility guarantees aiming to give the Ecosystem as a whole the following properties. See *Architecture* for the terms used here and how they fit together.

Application Portability

A Daml application should not depend on the underlying Database or DLT used by a Daml network.

Network Upgradeability

Ledger Operators should be able to upgrade Daml network or Participant Nodes seamlessly to stay up to date with the latest features and fixes. A Daml application should be able to operate without significant change across such Network Upgrades.

Daml Upgradeability

Application Developers should be able to update their developer tools seamlessly to stay up to date with the latest features and fixes, and stay able to maintain and develop their existing applications.

1.38.1 Ledger API Compatibility: Application Portability

Application Portability and to some extent Network Upgradeability are achieved by intermediating through the Ledger API. As per *Versioning*, and *Architecture*, the Ledger API is independently semantically versioned, and the compatibility guarantees derived from that semantic versioning extend to the entire semantics of the API, including the behavior of Daml Packages on the Ledger. Since all interaction with a Daml Ledger happens through the Daml Ledger API, a Daml Application is guaranteed to work as long as the Participant Node exposes a compatible Ledger API version.

Specifically, if a Daml Application is built against Ledger API version X.Y.Z and a Participant Node exposes Ledger API version X.Y2.Z2, the application is guaranteed to work as long as Y2.Z2 >= Y.Z.

Participant Nodes advertise the Ledger API version they support via the *version service*.

As a concrete example, Daml for Postgres 1.4.0 has the Participant Node integrated, and exposes Ledger API version 1.4.0 and the Daml for VMware Blockchain 1.0 Participant Nodes expose Ledger API version 1.6.0. So any application that runs on Daml for Postgres 1.4.0 will also run on Daml for VMware Blockchain 1.0.

1.38.1.1 List of Ledger API Versions Supported by Daml

The below lists with which Daml version a new Ledger API version was introduced.

Ledger API Version	Daml Version
2.4	2.7
2.3	2.6
2.2	2.5
2.1	2.4
2.0	2.0
1.12	1.15
1.11	1.14
1.10	1.11
1.9	1.10
1.8	1.9
<= 1.7	Introduced with the same Daml SDK version

1.38.1.2 Summary of Ledger API Changes

Ledger API Version	Changes
2.4	The IdentityProviderConfig record that contains the Identity Provider Config has been extended with an audience field. When set, the callers using JWT tokens issued by this identity provider are allowed to get an access only if the aud claim includes the string matching this specification. The identity_provider_id field on gRPC requests can be left empty if the JWT token submitted with the request already specifies an identity provider via an iss field. Users and parties can now be re-assigned between identity providers. The error codes and metadata of gRPC errors returned as part of failed command interpretation from the Ledger API have been updated to include more information. Previously, most errors from the Daml engine would be given as either GenericInterpretationError or InvalidArgumentInterpretationError. They now all have their own codes and encode relevant information in the gRPC Status metadata.
2.3	Introduce the Identity Provider Config Service. It makes possible for participant node administrators to setup and manage additional identity providers at runtime. This allows using access tokens from identity providers unknown at deployment time. When an identity provider is configured, independent IDP administrators can manage their own set of parties and users. Extend the Active Contract Service by adding *active_at_offset* field to the *GetActiveContractsRequest*. It defines an offset at which the snapshot of the active contracts will be computed. Extend the Metering Report Service by adding a JSON schema that defines the format of the reports in *GetMeteringReportResponse*. Extend the Transaction Service by adding a new *GetLatestPrunedOffsets* request. It allows querying for current pruning offsets. Introduce the Event Query Service. It allows querying for events associated with a given *ContractId* and *ContractKey*.
2.2	

1.38.2 Driver and Participant Compatibility: Network Upgradeability

Given the Ledger API Compatibility above, network upgrades are seamless if they preserve data, and Participant Nodes keep exposing the same or a newer minor version of the same major Ledger API Version. The semantic versioning of Daml drivers and participant nodes gives this guarantee. Upgrades from one minor version to another are data preserving, and major Ledger API versions may only be removed with a new major version of integration components, Daml drivers and Participant Nodes.

As an example, from an application standpoint, the only effect of upgrading Daml for Postgres 1.4.0 to Daml for Postgres 1.6.0 is an uptick in the Ledger API version. There may be significant changes to components or database schemas, but these are not public APIs.

1.38.2.1 Participant database migration

Participant Nodes automatically manage their database schema. The database schema is tied to the Daml version, and schema migrations are always data preserving. The below lists which Daml version can be upgraded from which Daml version.

Daml SDK version	Upgradeable from
2.1	1.7 or later
<= 2.0	1.0 or later

As an example, to upgrade a Participant Node built with Daml 1.4.0 to a version built with Daml 2.1, the operator should first upgrade to Daml 1.7 (or any other version between 1.7 and and 2.0), then upgrade to Daml 2.1.

1.38.3 SDK, Runtime Component, and Library Compatibility: Daml Upgradeability

As long as a major Ledger API version is supported (see *Ledger API Support Duration*), there will be supported version of Daml able to target all minor versions of that major version. This has the obvious caveat that new features may not be available with old Ledger API versions.

For example, an application built and compiled with Daml SDK 1.4.0 against Ledger API 1.4.0, it can still be compiled using SDK 1.6.0 and can be run against Ledger API 1.4.0 using 1.6.0 libraries and runtime components.

1.38.4 Ledger API Support Duration

Major Ledger API versions behave like stable features in *Status Definitions*. They are supported from the time they are first released as stable to the point where they are removed from Integration Components and Daml following a 12 month deprecation cycle. The earliest point a major Ledger API version can be deprecated is with the release of the next major version. The earliest it can be removed is 12 months later with a major version release of the Integration Components.

Other than for hotfix releases, new releases of the Integration Components will only support the latest minor/patch version of each major Ledger API version.

As a result we can make this overall statement:

An application built using Daml SDK U.V.W against Ledger API X.Y.Z can be maintained using any Daml SDK version U2.V2.W2 >= U.V.W as long as Ledger API major version X is still supported at the time of release of U2.V2.W2, and run against any Daml Network with Participant Nodes exposing Ledger API X.Y2.Z2 >= X.Y.Z.

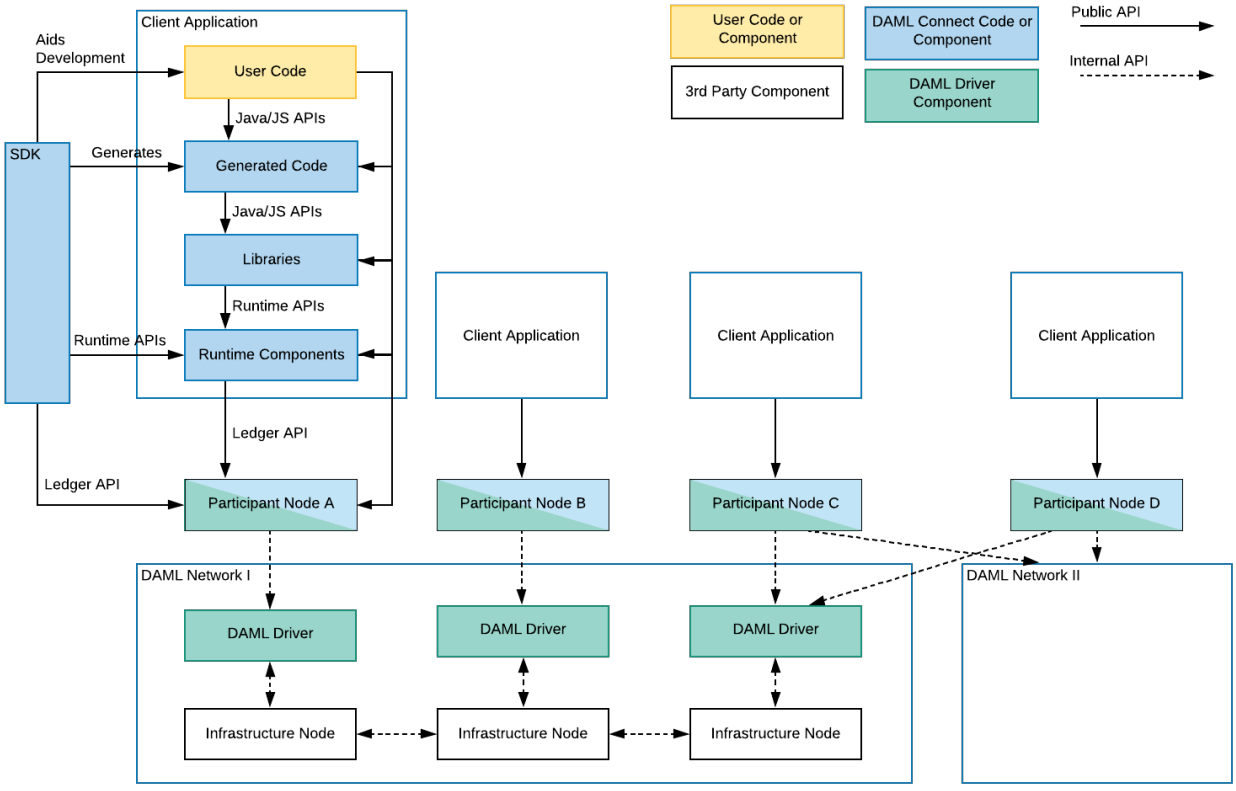
1.39 Daml Ecosystem Overview

This page is intended to give you an overview of the components that constitute the Daml Ecosystem, what status they are in, and how they fit together. It lays out Daml's public API in the sense of *Semantic Versioning*, and is a prerequisite to understanding Daml's *Portability, Compatibility, and Support Durations*.

The pages *Status Definitions* and *Feature and Component Statuses* give a fine-grained view of what labels like Alpha and Beta mean, which components expose public APIs and what status they are in.

1.39.1 Architecture

A high level view of the architecture of a Daml application or solution is helpful to make sense of how individual components, APIs and features fit into the Daml Stack.



The stack is segmented into two parts. Daml drivers encompass those components which enable an infrastructure to run Daml Smart Contracts, turning it into a **Daml Network**. **Daml Components** consists of everything developers and users need to connect to a Daml Network: the tools to build, deploy, integrate, and maintain a Daml Application.

Taking the diagram from left to right, the SDK acts on various components of the client application and directly on the participant nodes: it aids in the development of user code, generates code of its own, feeds into runtime components via runtime APIs, and creates participant nodes via the Ledger API. The client application also acts on participant nodes via the Ledger API, and the user code for that application can act on the various Daml components of the application (generated code, libraries, and runtime components) via public API. Participant nodes, in turn, act via an internal API on the Daml network, specifically with Daml drivers that in turn interact with infrastructure nodes. The infrastructure nodes can also interact with each other. Each client application is linked to only one participant node, but a participant node can potentially touch more than one Daml network.

1.39.2 Daml Networks

1.39.2.1 Daml drivers

At the bottom of every Daml Application is a Daml network, a distributed, or possibly centralized persistence infrastructure together with Daml drivers. Daml drivers enable the persistence infrastructure to act as a consensus, messaging, and in some cases persistence layer for Daml Applications. Most Daml drivers will have a public API, but there are no *uniform* public APIs on Daml drivers. This does not harm application portability since applications only interact with Daml networks through the Participant Node. A good example of a public API of a Daml driver is the deployment interface of Daml for VMware Blockchain. It's a public interface, but specific to the VMware driver.

1.39.3 Participant Nodes

On top of, or integrated into the Daml drivers sits a Participant Node, that has the primary purpose of exposing the Daml Ledger API. In the case of *integrated* Daml drivers, the Participant Node usually interacts with the Daml drivers through solution-specific APIs. In this case, Participant Nodes can only communicate with Daml drivers of one Daml Network. In the case of *interoperable* Daml drivers, the Participant Node communicates with the Daml drivers through the uniform Canton Protocol. The Canton Protocol is versioned and has some cross-version compatibility guarantees, but is not a public API. So participant nodes may have public APIs like monitoring and logging, command line interfaces or similar, but the only *uniform* public API exposed by all Participant Nodes is the Ledger API.

1.39.4 Ledger API

The Ledger API is the primary interface that offers forward and backward compatibility between Daml Networks and Applications (including Daml components). As you can see in the diagram above, all interaction between components above the Participant Node and the Participant Node or Daml Network happen through the Ledger API. The Ledger API is a public API and offers the lowest level of access to Daml Ledgers supported for application use.

1.39.5 Daml Components

1.39.5.1 Runtime Components

Runtime components are standalone components that run alongside Participant Nodes or Applications and expose additional services like query endpoints, automations, or integrations. Each Runtime Component has public APIs, which are covered in *Feature and Component Statuses*. Typically there is a command line interface, and one or more Runtime APIs as indicated in the above diagram.

1.39.5.2 Libraries

Libraries naturally provide public APIs in their target language, be it Daml, or secondary languages like JavaScript or Java. For details on available libraries and their interfaces, see *Feature and Component Statuses*.

1.39.5.3 Generated Code

The SDK allows the generation of code for some languages from a Daml Model. This generated code has public APIs, which are not independently versioned, but depend on the Daml version and source of the generated code, like a Daml package. In this case, the version of the Daml SDK used covers changes to the public API of the generated code.

1.39.5.4 Developer Tools / SDK

The Daml SDK consists of the developer tools used to develop user code, both Daml and in secondary languages, to generate code, and to interact with running applications via Runtime, and Ledger API. The SDK has a broad public API covering the Daml Language, CLIs, IDE, and Developer tools, but few of those APIs are intended for runtime use in a production environment. Exceptions to that are called out on *Feature and Component Statuses*.

1.39.6 Status Definitions

Throughout the documentation, we use labels to mark features of APIs not yet deemed stable. This page gives meaning to those labels.

1.39.6.1 Early Access Features

Features or components covered by these docs are *Stable* by default. *Stable* features and components constitute Daml's public API in the sense of *Semantic Versioning*. Feature and components that are not *Stable* are called Early Access and called out explicitly.

Early Access features are opt-in whenever possible, needing to be activated with special commands or flags needing to be started up separately, or requiring the use of additional endpoints, for example.

Within the Early Access category, we distinguish three labels:

Labs

Labs components and features are experiments, introduced for evaluation, testing, or project-internal use. There is no intent to develop them into a stable feature other than to see whether they add value and find uptake. They can be changed or discontinued without advance notice. They may be poorly documented and it is not recommended to start relying on them.

Alpha

Alpha components and features are early preview versions of features being actively developed to become a stable part of the ecosystem. At the Alpha stage, they are not yet feature complete, may have poor runtime characteristics, are still subject to frequent change, and may not be fully documented. Alpha features can be evaluated, and used in PoCs, but should not yet be relied upon for large projects or production use where breakages or changes to APIs would be costly.

Beta

Beta components and features are preview versions of features that are close to maturity. They are characterized by being considered feature complete, and the APIs close to the final public APIs. It is relatively safe to build on Beta features as long as the documented caveats to runtime characteristics are understood and bugs and minor API adjustments are not too costly.

1.39.6.2 Deprecation

In addition to being labelled Early Access, features and components can also be labelled Deprecated . Deprecation follows a deprecation cycle laid out in the table below. The date of deprecation is documented in *Daml Ecosystem Overview*.

Deprecated features can be relied upon during the deprecation cycle to the same degree as their non-deprecated counterparts, but building on deprecated features may hinder an upgrade to new Daml versions following the deprecation cycle.

1.39.6.3 Comparison of Statuses

The table below gives a concise overview of the labels used for Daml features and components.

Table 4: Feature Maturities

	Stable	Beta	Alpha	Labs
Functionality				
Functional Completeness	Functionally complete	Considered functionally complete, but subject to change according to usability testing	MVP-level functionality covering at least a few core use-cases	Functionality covering one specific use-case it was made for
Non-functional Requirements				
Performance	Unless stated otherwise, the feature can be used without concern about system performance.	Current performance impacts and expected performance for the stable release are documented.	Using the feature may have significant undocumented impact on overall system performance.	Using the feature may have significant undocumented impact on overall system performance.
Compatibility	Compatibility is covered by *Portability, Compatibility, and Support Durations*.	Compatibility is covered by *Portability, Compatibility, and Support Durations*.	The feature may only work against specific Daml integrations, or specific API versions, including Early Access ones.	The feature may only work against specific Daml integrations, or specific API versions, including Early Access ones.
Stability & Error Recovery	The feature is long-term stable and supports recovery fit for a production system.	No known reproducible crashes which can't be recovered from. There is still an expectation that new issues may be discovered.	The feature may not be stable and lack error recovery.	The feature may not be stable and lack error recovery.
Releases and Support				
Distribution and Releases	Distributed as part of regular *releases*.	Distributed as part of regular *releases*.	Distributed as part of regular *releases*.	Releases and distribution may be separate.
Support	Covered by standard commercial support terms. Hotfixes for critical bugs and security issues are available.	Not covered by standard commercial support terms. Receives bug- and security fixes with regular releases.	Not covered by standard commercial support terms. Receives bug- and security fixes with regular releases.	Not covered by standard commercial support terms. Only receives fixes with low priority.
Deprecation	May be removed with any new major version 12 months after the date of deprecation.	May be removed with any new minor version 1 month after the date of deprecation.	May be removed without warning.	May be removed without warning.
Covered	Yes, part of the	No, but breaking	No, and changes	No, and changes

1.39.7 Feature and Component Statuses

This page gives an overview of the statuses of released components and features according to *Status Definitions*. Anything not listed here implicitly has status Labs , but it's possible that something accidentally slipped the list so if in doubt, please *contact us*.

1.39.7.1 Ledger API

Component/Feature	Status	Deprecated on
Ledger API specification including all semantics of >= Daml-LF 1.6	Stable	
Numbered (ie non-dev) Versions of Proto definitions distributed via GitHub Releases	Stable	
Dev Versions of Proto definitions distributed via GitHub Releases	Alpha	
Use of divulged contracts in later transactions	Stable, Deprecated	2021-06-16

1.39.7.2 Runtime Components

Component / Feature	Status	Deprecated on
Canton		
Canton Application and Console	Stable	
Canton Administrative APIs for participant and domain nodes	Stable	
Canton Protocol	Stable	
Sequencer for PostgreSQL	Stable	
Sequencer for Oracle DB	Stable	
Sequencer for Hyperledger Fabric	Beta	
Sequencer for Hyperledger Besu	Beta	
Support for connecting a single participant to multiple domains	Alpha	
JSON API		
HTTP endpoints under `/v1/` including status codes, authentication, query language and encoding.	Stable	
`daml json-api` CLI *for development*. (as specified using `daml json-api --help`)	Stable	
Stand-alone distribution for production use, including CLI specified in `--help`.	Stable	
Triggers		
Daml API of individual Triggers	Stable	
Development CLI to start individual triggers in dev environment (`daml trigger`)	Stable	
Trigger Service (`daml trigger-service`)	Stable	
Non-repudiation		
Non-repudiation	Alpha	

1.39.7.3 Libraries

Component / Feature	Status	Deprecated on
Java Ledger API Bindings		
`daml codegen java` *CLI and generated code*	Stable	
`bindings-java` *library and its public API*.	Stable	
`bindings-rxjava` *library and its public API*.	Stable	
`daml-lf-1.6-archive-java-proto`	Stable	
`daml-lf-1.7-archive-java-proto`	Stable	
`daml-lf-1.8-archive-java-proto`	Stable	
`daml-lf-dev-archive-java-proto`	Alpha	
Python Ledger API Bindings (formerly known as DAZL)		
`dazl` library and its public API	Stable	
JavaScript Client Libraries		
`daml codegen js` *CLI and generated code*	Stable	
`@daml/types` *library and its public API*	Stable	
`@daml/ledger` *library and its public API*	Stable	
`@daml/react` *library and its public API*	Stable	
Daml Libraries		
The Daml Standard Library	Stable	
The Daml Script Library	Stable	
The Daml Trigger Library	Stable	

1.39.7.4 Developer Tools

Component / Feature	Status	Deprecated on
SDK		
Windows SDK	Stable	
Mac SDK	Stable	
Linux SDK	Stable	

Daml Assistant (daml) with top level commands
`--help`
`version`
`install`
`uninstall`
	Stable	

`daml start` helper command and associated CLI (`daml start --help`)	Stable	
`daml deploy` *helper command and associated CLI* (`daml deploy --help`)	Stable	

continues on next page

Table 5 – continued from previous page

Component / Feature	Status	Deprecated on
Assistant commands to start Runtime Components: `daml json-api`, `daml trigger`, and `daml trigger-service`.	See *Runtime Components*.	
Daml Projects		
`daml.yaml` project specification	Stable	
Assistant commands `new`, `create-daml-app`, and `init`. Note that the templates created by `daml new` and `create-daml-app` are considered example code, and are not covered by *semantic versioning*.	Stable	
Daml Studio		
VSCode Extension	Stable	
`daml studio` assistant command	Stable	
Code Generation		
`daml codegen` assistant commands	See *Libraries*.	
Sandbox Development Ledger		
`daml sandbox` assistant command and documented CLI under `daml sandbox --help`.	Stable	
Daml Profiler in Sandbox	Stable	
Daml Compiler		
`daml build` CLI	Stable	
`daml damlc` CLI	Stable	
Compilation and packaging (`daml damlc build`)	Stable	
Legacy packaging command (`daml damlc package`)	Stable, Deprecated	2020-10-14
In-memory Scenario/Script testing (`daml damlc test`)	Stable	
DAR File inspection (`daml damlc inspect-dar`). The exact output is only covered by *semantic versioning* when used with the `--json` flag.	Stable	
DAR File validation (`daml damlc validate-dar`)	Stable	
Daml Linter (`daml damlc lint`)	Stable	
Daml REPL (`daml damlc repl`)	See Daml REPL heading below	
Daml Language Server CLI (`daml damlc ide`)	Labs	
Daml Documentation Generation (`daml damlc docs`)	Labs	
`daml doctest`	Labs	
Script		
Script Daml API	Stable	
Daml Scenario IDE integration	Stable	

continues on next page

Table 5 – continued from previous page

Component / Feature	Status	Deprecated on
Daml Script IDE integration	Stable	
Daml Script Library	See *Libraries*	
`daml test` in-memory Script and Scenario test CLI	Stable	
`daml script` *CLI to run Scripts against live ledgers.*	Stable	
Navigator		
Daml Navigator Development UI (`daml navigator server`)	Stable	
Navigator Config File Creation (`daml navigator create-config`)	Stable	
Navigator graphQL Schema (`daml navigator dump-graphql-schema`)	Labs	
Daml REPL Interactive Shell		
`daml repl` *CLI*	Stable	
Daml and meta-APIs of the REPL	Stable	
Ledger Administration CLI		
`daml ledger` *CLI and all subcommands*.	Stable	

1.40 Releases and Versioning

1.40.1 Versioning

All Daml components follow Semantic Versioning. In short, this means that there is a well defined public API , changes or breakages to which are indicated by the version number.

Stable releases have versions MAJOR.MINOR.PATCH. Segments of the version are incremented according to the following rules:

1. MAJOR version when there are incompatible API changes,
2. MINOR version when functionality is added in a backwards compatible manner, and
3. PATCH version when there are only backwards compatible bug fixes.

Daml's public API is laid out in the *Daml Ecosystem Overview*.

1.40.2 Cadence

Regular, weekly snapshot releases are made every Wednesday, with additional snapshots produced as needed. These releases contain Daml Components, both from the daml repository as well as some others.

The decision to perform a Minor version release is based on the content or scope of the payload of that release. The intent is to release a Minor version once a quarter but this may change based on the customer demand for new, key features.

No more than one major version is released every six months, barring exceptional circumstances.

Individual Daml drivers follow their own release cadence, using already released Integration Components as a dependency.

1.40.3 Support Duration

Major versions will be supported for a minimum of one year after a subsequent Major version is release. Within a major version, only the latest minor version receives security and bug fixes.

1.40.4 Release Notes

Release notes for each release are published on the Release Notes section of the Daml Driven blog.

1.40.5 Process

Weekly snapshot and Minor releases follow a common process. The process is documented in the Daml repository. Only the schedule for Minor releases is covered below.

Selecting a Release Candidate

> This is done by the Daml core engineering teams.
>
> The Minor releases are scope-based. Furthermore, Daml development is fully HEAD-based so both the repository and every snapshot are intended to be in a fully releasable state at every point. The release process therefore starts with selecting a release candidate. Typically the Snapshot from the preceding Wednesday is selected as the release candidate.

Release Notes and Candidate Review

> After selecting the release candidate, Release Notes are written and reviewed with a particular view towards unintended changes and violations of *Semantic Versioning*.

Release Candidate Refinement

> If issues surface in the initial review, the issues are resolved and different Snapshot is selected as the release candidate.

Release Candidate Announcement

> Barring delays due to issues during initial review, the release candidate is announced publicly with accompanying Release Notes.

Communications, Testing and Feedback

> In the days following the announcement, the release is presented and discussed with both commercial and community users. It is also put through its paces by integrating it in Daml Hub and several ledger integrations.

Release Candidate Refinement II

> Depending on feedback and test results, new release candidates may be issued iteratively. Depending on the severity of changes from release candidate to release candidate, the testing period is extended more or less.

Release

> Assuming the release is not postponed due to extended test periods or newly discovered issues in the release candidate, the release is declared stable and given a regular version number.

1.41 Glossary of concepts

1.41.1 Key Concepts

1.41.1.1 Daml

Daml is a platform for building and running sophisticated, multi-party applications. At its core, it contains a smart contract *language* and *tooling* that defines the schema, semantics, and execution of transactions between parties. Daml includes *Canton*, a privacy-enabled distributed ledger that is enhanced when deployed with complementary blockchains.

1.41.1.2 Daml Language

The Daml language is a purpose-built language for rapid development of composable multi-party applications. It is a modern, ergonomically designed functional language that carefully avoids many of the pitfalls that hinder multi-party application development in other languages.

1.41.1.3 Daml Ledger

A Daml ledger is a distributed ledger system running *Daml smart contracts* according to the *Daml ledger model* and exposes the Daml Ledger APIs. All current implementations of Daml ledgers consist of a Daml driver that utilizes an underlying Synchronization Technology to either implement the Daml ledger directly, or to run the Canton protocol.

Canton Ledger

A Canton ledger is a privacy-enabled Daml ledger implemented using the Canton application, nodes, and protocol.

1.41.1.4 Canton Protocol

The Canton protocol is the technology which synchronizes *participant nodes* across any Daml-enabled blockchain or database. The Canton protocol not only makes Daml applications portable between different underlying *synchronization technologies*, but also allows applications to transact with each other across them.

1.41.1.5 Synchronization Technology

The synchronization technology is the database or blockchain that Daml uses for synchronization, messaging, and topology. Daml runs on a range of synchronization technologies, from centralized databases to fully distributed deployments, and users can employ the technology that best suits their technical and operational needs.

1.41.1.6 Daml Drivers

Daml drivers enable a *ledger* to be implemented on top of different *synchronization technologies*; a database or distributed ledger technology.

1.41.2 Daml Language Concepts

1.41.2.1 Contract

Contracts are items on a *ledger*. They are created from blueprints called *templates*, and include:

data (parameters)
roles (*signatory*, *observer*)
choices (and *controllers*)

Contracts are immutable: once they are created on the ledger, the information in the contract cannot be changed. The only thing that can happen to them is that they can be *archived*.

Active Contract, Archived Contract

When a *contract* is created on a *ledger*, it becomes **active**. But that doesn't mean it will remain active forever: it can be **archived**. This can happen:

if the *signatories* of the contract decide to archive it
if a *consuming choice* is exercised on the contract

Once the contract is archived, it is no longer valid, and *choices* on the contract can no longer be exercised.

1.41.2.2 Template

A **template** is a blueprint for creating a *contract*. This is the Daml code you write.

For full documentation on what can be in a template, see *Reference: Templates*.

1.41.2.3 Choice

A **choice** is something that a *party* can *exercise* on a *contract*. You write code in the choice body that specifies what happens when the choice is exercised: for example, it could create a new contract.

Choices give one a way to transform the data in a contract: while the contract itself is immutable, you can write a choice that *archives* the contract and creates a new version of it with updated data.

A choice can only be exercised by its *controller*. Within the choice body, you have the *authorization* of all of the contract's *signatories*.

For full documentation on choices, see *Reference: Choices*.

Consuming Choice

A **consuming choice** means that, when the choice is exercised, the *contract* it is on will be *archived*. The alternative is a *nonconsuming choice*.

Consuming choices can be *preconsuming* or *postconsuming*.

Preconsuming Choice

A *choice* marked **preconsuming** will be *archived* at the start of that *exercise*.

Postconsuming Choice

A *choice* marked **postconsuming** will not be *archived* until the end of the *exercise* choice body.

Nonconsuming Choice

A **nonconsuming choice** does NOT *archive* the *contract* it is on when *exercised*. This means the choice can be exercised more than once on the same *contract*.

Disjunction Choice, Flexible Controllers

A **disjunction choice** has more than one *controller*.

If a contract uses **flexible controllers**, this means you don't specify the controller of the *choice* at *creation* time of the *contract*, but at *exercise* time.

1.41.2.4 Party

A **party** represents a person or legal entity. Parties can *create contracts* and *exercise choices*.

***Signatories*, *observers*, *controllers*, and *maintainers* all must be parties, represented by the `Party` data type i**
contract data.

Parties are hosted on participant nodes and a participant node can host more than one party. A party can be hosted on several participant nodes simultaneously.

Signatory

A **signatory** is a *party* on a *contract*. The signatories MUST consent to the *creation* of the contract by *authorizing* it: if they don't, contract creation will fail. Once the contract is created, signatories can see the contracts and all exercises of that contract.

For documentation on signatories, see *Reference: Templates*.

Observer

An **observer** is a *party* on a *contract*. Being an observer allows them to see that instance and all the information about it. They do NOT have to *consent to* the creation.

For documentation on observers, see *Reference: Templates*.

Controller

A **controller** is a *party* that is able to *exercise* a particular *choice* on a particular *contract*.

Controllers must be at least an *observer*, otherwise they can't see the contract to exercise it on. But they don't have to be a *signatory*. this enables the *propose-accept pattern*.

Choice Observer

A **choice observer** is a *party* on a *choice*. Choice observers are guaranteed to see the choice being exercised and all its consequences with it.

Stakeholder

Stakeholder is not a term used within the Daml language, but the concept refers to the *signatories* and *observers* collectively. That is, it means all of the *parties* that are interested in a *contract*.

Maintainer

The **maintainer** is a *party* that is part of a *contract key*. They must always be a *signatory* on the *contract* that they maintain the key for.

It's not possible for keys to be globally unique, because there is no party that will necessarily know about every contract. However, by including a party as part of the key, this ensures that the maintainer *will* know about all of the contracts, and so can guarantee the uniqueness of the keys that they know about.

For documentation on contract keys, see *Reference: Contract Keys*.

1.41.2.5 Authorization, Signing

The Daml runtime checks that every submitted transaction is **well-authorized**, according to the *authorization rules of the ledger model*, which guarantee the integrity of the underlying ledger.

A Daml update is the composition of update actions created with one of the items in the table below. A Daml update is well-authorized when **all** its contained update actions are well-authorized. Each operation has an associated set of parties that need to authorize it:

Table 6: Updates and required authorization

Update action	Type	Authorization
`create`	`(Template c) => c -> Update (ContractId c)`	All signatories of the created contract
`exercise`	`ContractId c -> e -> Update r`	All controllers of the choice
`fetch`	`ContractId c -> e -> Update r`	One of the union of signatories and observers of the fetched contract
`fetchByKey`	`k -> Update (ContractId c, c)`	Same as `fetch`
`lookupByKey`	`k -> Update (Optional (ContractId c))`	All key maintainers

At runtime, the Daml execution engine computes the required authorizing parties from this mapping. It also computes which parties have given authorization to the update in question. A party gives authorization to an update in one of two ways:

It is the signatory of the contract that contains the update action.
It is an element of the controllers executing the choice containing the update action.

Only if all required parties have given their authorization to an update action, the update action is well-authorized and therefore executed. A missing authorization leads to the abortion of the update action and the failure of the containing transaction.

It is noteworthy, that authorizing parties are always determined only from the local context of a choice in question, that is, its controllers and the contract's signatories. Authorization is never inherited from earlier execution contexts.

1.41.2.6 Standard Library

The **Daml standard library** is a set of *Daml* functions, classes and more that make developing with Daml easier.

For documentation, see *The standard library*.

1.41.2.7 Agreement

An **agreement** is part of a *contract*. It is the text that explains what the contract represents.

It can be used to clarify the legal intent of a contract, but this text isn't evaluated programmatically.

See *Reference: Templates*.

1.41.2.8 Create

A **create** is an update that creates a *contract* on the *ledger*.

Contract creation requires *authorization* from all its *signatories*, or the create will fail. For how to get authorization, see the *propose-accept* and *multi-party agreement* patterns.

A *party submits* a create *command*.

See *Reference: Updates*.

1.41.2.9 Exercise

An **exercise** is an action that exercises a *choice* on a *contract* on the *ledger*. If the choice is *consuming*, the exercise will *archive* the contract; if it is *nonconsuming*, the contract will stay active.

Exercising a choice requires *authorization* from all of the *controllers* of the choice.

A *party submits* an exercise *command*.

See *Reference: Updates*.

1.41.2.10 Daml Script

Daml Script provides a way of testing Daml code during development. You can run Daml Script inside *Daml Studio*, or write them to be executed on *Sandbox* when it starts up.

They're useful for:

expressing clearly the intended workflow of your *contracts*
ensuring that parties can exclusively create contracts, observe contracts, and exercise choices that they are meant to
acting as regression tests to confirm that everything keeps working correctly

In Daml Studio, Daml Script runs in an emulated ledger. You specify a linear sequence of actions that various parties take, and these are evaluated in order, according to the same consistency, authorization, and privacy rules as they would be on a Daml ledger. Daml Studio shows you the resulting *transaction* graph, and (if a Daml Script fails) what caused it to fail.

See *Test Templates Using Daml Script*.

1.41.2.11 Contract Key

A **contract key** allows you to uniquely identify a *contract* of a particular *template*, similar to a primary key in a database table.

A contract key requires a *maintainer*: a simple key would be something like a tuple of text and maintainer, like `(accountId, bank)`.

See *Reference: Contract Keys*.

1.41.2.12 DAR File, DALF File

A Daml Archive file, known as a `.dar` file is the result of compiling Daml code using the *Assistant* which can be interpreted using a Daml interpreter.

You upload `.dar` files to a *ledger* in order to be able to create contracts from the templates in that file.

A `.dar` contains multiple `.dalf` files. A `.dalf` file is the output of a compiled Daml package or library. Its underlying format is *Daml-LF*.

1.41.3 Developer Tools

1.41.3.1 Assistant

Daml Assistant is a command-line tool for many tasks related to Daml. Using it, you can create Daml projects, compile Daml projects into *.dar files*, launch other developer tools, and download new SDK versions.

See *Daml Assistant (daml)*.

1.41.3.2 Studio

Daml Studio is a plugin for Visual Studio Code, and is the IDE for writing Daml code.

See *Daml Studio*.

1.41.3.3 Sandbox

Sandbox is a lightweight ledger implementation. In its normal mode, you can use it for testing.

You can also run the Sandbox connected to a PostgreSQL back end, which gives you persistence and a more production-like experience.

See *Daml Sandbox*.

1.41.3.4 Navigator

Navigator is a tool for exploring what's on the ledger. You can use it to see what contracts can be seen by different parties, and *submit commands* on behalf of those parties.

Navigator GUI

This is the version of Navigator that runs as a web app.

See *Navigator*.

1.41.4 Building Applications

1.41.4.1 Application, Ledger Client, Integration

Application, **ledger client**, and **integration** are all terms for an application that sits on top of the *ledger*. These usually *read from the ledger*, *send commands* to the ledger, or both.

There's a lot of information available about application development, starting with the *Daml Application Architecture* page.

1.41.4.2 Ledger API

The **Ledger API** is an API that's exposed by any *ledger* on a participant node. Users access and manipulate the ledger state through the Ledger API. An alternative name for the Ledger API is the **gRPC Ledger API** if disambiguation from other technologies is needed. See *The Ledger API* page. It includes the following *services*.

Command Submission Service

Use the **command submission service** to *submit commands* - either create commands or exercise commands - to the *ledger*. See *Command Submission Service*.

Command Completion Service

Use the **command completion service** to find out whether or not *commands you have submitted* have completed, and what their status was. See *Command Completion Service*.

Command Service

Use the **command service** when you want to *submit a command* and wait for it to be executed. See *Command Service*.

Transaction Service

Use the **transaction service** to listen to changes in the *ledger*, reported as a stream of *transactions*. See *Transaction Service*.

Active Contract Service

Use the **active contract service** to obtain a party-specific view of all *contracts* currently *active* on the *ledger*. See *Active Contracts Service*.

Package Service

Use the **package service** to obtain information about Daml packages available on the *ledger*. See *Package Service*.

Ledger Identity Service

Use the **ledger identity service** to get the identity string of the *ledger* that your application is connected to. See *Ledger Identity Service (DEPRECATED)*.

Ledger Configuration Service

Use the **ledger configuration service** to subscribe to changes in *ledger* configuration. See *Ledger Configuration Service*.

1.41.4.3 Ledger API Libraries

The following libraries wrap the *ledger API* for more native experience applications development.

Java Bindings

An idiomatic Java library for writing *ledger applications*. See *Java Bindings*.

Python Bindings

A Python library (formerly known as DAZL) for writing *ledger applications*. See *Python Bindings*.

1.41.4.4 Reading From the Ledger

Applications get information about the *ledger* by **reading** from it. You can't query the ledger, but you can subscribe to the transaction stream to get the events, or the more sophisticated active contract service.

1.41.4.5 Submitting Commands, Writing To the Ledger

Applications make changes to the *ledger* by **submitting commands**. You can't change it directly: an application submits a command of *transactions*. The command gets evaluated by the runtime, and will only be accepted if it's valid.

For example, a command might get rejected because the transactions aren't *well-authorized*; because the contract isn't *active* (perhaps someone else archived it); or for other reasons.

This is echoed in *Daml script*, where you can mock an application by having parties submit transactions/updates to the ledger. You can use `submit` or `submitMustFail` to express what should succeed and what shouldn't.

Commands

A **command** is an instruction to add a transaction to the *ledger*.

1.41.4.6 Participant Node

The participant node is a server that provides users with consistent programmatic access to a ledger through the *Ledger API*. The participant nodes handle transaction signing and validation, such that users don't have to deal with cryptographic primitives but can trust the participant node that the data they are observing has been properly verified to be correct.

1.41.4.7 Sub-transaction Privacy

Sub-transaction privacy is where participants in a transaction only learn about the subset of the transaction they are directly involved in, but not about any other part of the transaction. This applies to both the content of the transaction as well as other involved participants.

1.41.4.8 Daml-LF

When you compile Daml source code into a *.dar file*, the underlying format is **Daml-LF**. Daml-LF is similar to Daml, but is stripped down to a core set of features. The relationship between the surface Daml syntax and Daml-LF is loosely similar to that between Java and JVM bytecode.

As a user, you don't need to interact with Daml-LF directly. But internally, it's used for:

executing Daml code on the Sandbox or on another platform
sending and receiving values via the Ledger API (using a protocol such as gRPC)
generating code in other languages for interacting with Daml models (often called codegen)

1.41.4.9 Composability

Composability is the ability of a participant to extend an existing system with new Daml applications or new topologies unilaterally without requiring cooperation from anyone except the directly involved participants who wish to be part of the new application functionality.

1.41.4.10 Trust Domain

A trust domain encompasses a part of the system (in particular, a Daml ledger) operated by a single real-world entity. This subsystem may consist of one or more physical nodes. A single physical machine is always assumed to be controlled by exactly one real-world entity.

1.41.5 Canton Concepts

1.41.5.1 Domain

The domain provides total ordered, guaranteed delivery multi-cast to the participants. This means that participant nodes communicate with each other by sending end-to-end encrypted messages through the domain.

The *sequencer service* of the domain orders these messages without knowing about the content and ensures that every participant receives the messages in the same order.

The other services of the domain are the *mediator* and the *domain identity manager*.

1.41.5.2 Private Contract Store

Every participant node manages its own private contract store (PCS) which contains only contracts the participant is privy to. There is no global state or global contract store.

1.41.5.3 Virtual Global Ledger

While every participant has their own private contract store (PCS), the *Canton protocol* guarantees that the contracts which are stored in the PCS are well-authorized and that any change to the store is justified, authorized, and valid. The result is that every participant only possesses a small part of the *virtual global ledger*. All the local stores together make up that *virtual global ledger* and they are thus synchronized. The Canton protocol guarantees that the virtual ledger provides integrity, privacy, transparency, and auditability. The ledger is logically global, even though physically, it runs on segregated and isolated domains that are not aware of each other.

1.41.5.4 Mediator

The mediator is a service provided by the *domain* and used by the *Canton protocol*. The mediator acts as commit coordinator, collecting individual transaction verdicts issued by validating participants and aggregating them into a single result. The mediator does not learn about the content of the transaction, they only learn about the involved participants.

1.41.5.5 Sequencer

The sequencer is a service provided by the *domain*, used by the *Canton protocol*. The sequencer forwards encrypted addressed messages from participants and ensures that every member receives the messages in the same order. Think about registered and sealed mail delivered according to the postal datestamp.

1.41.5.6 Domain Identity Manager

The Domain Identity Manager is a service provided by the *domain*, used by the *Canton protocol*. Participants join a new domain by registering with the domain identity manager. The domain identity manager establishes a consistent identity state among all participants. The domain identity manager only forwards identity updates. It can not invent them.

1.41.5.7 Consensus

The Canton protocol does not use PBFT or any similar consensus algorithm. There is no proof of work or proof of stake involved. Instead, Canton uses a variant of a stakeholder-based two-phase commit protocol. As such, only stakeholders of a transaction are involved in it and need to process it, providing efficiency, privacy, and horizontal scalability. Canton-based ledgers are resilient to malicious participants as long as there is at least a single honest participant. A domain integration itself might be using the consensus mechanism of the underlying platform, but participant nodes will not be involved in that process.

1.42 Daml Example Applications

Click to open our collection of

1.43 Daml Language References

1.43.1 Daml Language Cheat Sheet

Click to open the

1.43.2 Language Reference

This section contains a reference to writing templates for Daml contracts. It includes:

1.43.2.1 Overview: Template Structure

This page covers what a template looks like: what parts of a template there are, and where they go.

For the structure of a Daml file *outside* a template, see *Reference: Daml File Structure*.

Template Outline Structure

Here's the structure of a Daml template:

```
template NameOfTemplate
  with
    exampleParty : Party
    exampleParty2 : Party
    exampleParty3 : Party
    exampleParameter : Text
    -- more parameters here
  where
    signatory exampleParty
    observer exampleParty2
    agreement
      -- some text
      ""
    ensure
      -- boolean condition
      True
    key (exampleParty, exampleParameter) : (Party, Text)
    maintainer (exampleFunction key)
    -- a choice goes here; see next section
```

template name `template` keyword
parameters `with` followed by the names of parameters and their types
template body `where` keyword
Can include:
template-local definitions `let` keyword
Lets you make definitions that have access to the contract arguments and are available in the rest of the template definition.
signatories `signatory` keyword
Required. The parties (see the *Party* type) who must consent to the creation of this contract. You won't be able to create this contract until all of these parties have authorized it.
observers `observer` keyword
Optional. Parties that aren't signatories but who you still want to be able to see this contract.
an agreement `agreement` keyword
Optional. Text that describes the agreement that this contract represents.
a precondition `ensure` keyword
Only create the contract if the conditions after `ensure` evaluate to true.
a contract key `key` keyword
Optional. Lets you specify a combination of a party and other data that uniquely identifies a contract of this template. See *Reference: Contract Keys*.
maintainers `maintainer` keyword
Required if you have specified a `key`. Keys are only unique to a `maintainer`. See *Reference: Contract Keys*.
choices `choice NameOfChoice : ReturnType controller nameOfParty do`
or
`controller nameOfParty can NameOfChoice : ReturnType do`
Defines choices that can be exercised. See *Choice structure* for what can go in a choice. Note that `controller`-first syntax is deprecated and will be removed in a future version of Daml.

Choice Structure

Here's the structure of a choice inside a template. There are two ways of specifying a choice:

start with the `choice` keyword
start with the `controller` keyword

```
-- option 1 for specifying choices: choice name first
choice NameOfChoice
  : () -- replace () with the actual return type
  with
    party : Party -- parameters here
  controller party
  do
    return () -- replace this line with the choice body

-- option 2 for specifying choices (deprecated syntax): controller first
controller exampleParty can
  NameOfAnotherChoice
    : () -- replace () with the actual return type
    with
      party : Party -- parameters here
    do
      return () -- replace the line with the choice body
```

a controller (or controllers) `controller` keyword
Who can exercise the choice.

choice observers `observer` keyword
Optional. Additional parties that are guaranteed to be informed of an exercise of the choice.
To specify choice observers, you must start you choice with the `choice` keyword.
The optional `observer` keyword must precede the mandatory `controller` keyword.

consumption annotation Optionally one of `preconsuming`, `postconsuming`, `nonconsuming`, which changes the behavior of the choice with respect to privacy and if and when the contract is archived. See *contract consumption in choices* for more details.

a name Must begin with a capital letter. Must be unique - choices in different templates can't have the same name.

a return type after a `:`, the return type of the choice

choice arguments `with` keyword
If you start your choice with `choice` and include a `Party` as a parameter, you can make that `Party` the `controller` of the choice. This is a feature called flexible controllers , and it means you don't have to specify the controller when you create the contract - you can specify it when you exercise the choice. To exercise a choice, the party needs to be a signatory or an observer of the contract and must be explicitly declared as such.

a choice body After `do` keyword
What happens when someone exercises the choice. A choice body can contain update statements: see *Choice body structure* below.

Choice Body Structure

A choice body contains `Update` expressions, wrapped in a *do* block.

The update expressions are:

create Create a new contract of this template.
`create NameOfContract with contractArgument1 = value1; contractArgument2 = value2; ...`

exercise Exercise a choice on a particular contract.
`exercise idOfContract NameOfChoiceOnContract with choiceArgument1 = value1; choiceArgument2 = value 2; ...`

fetch Fetch a contract using its ID. Often used with assert to check conditions on the contract's content.
`fetchedContract <- fetch IdOfContract`

fetchByKey Like `fetch`, but uses a *contract key* rather than an ID.
`fetchedContract <- fetchByKey @ContractType contractKey`

lookupByKey Confirm that a contract with the given *contract key* exists.
`fetchedContractId <- lookupByKey @ContractType contractKey`

abort Stop execution of the choice, fail the update.
`if False then abort`

assert Fail the update unless the condition is true. Usually used to limit the arguments that can be supplied to a contract choice.
`assert (amount > 0)`

getTime Gets the ledger time. Usually used to restrict when a choice can be exercised.
`currentTime <- getTime`

return Explicitly return a value. By default, a choice returns the result of its last update expression. This means you only need to use `return` if you want to return something else.
`return ContractID ExampleTemplate`

The choice body can also contain:

***let* keyword** Used to assign values or functions.

assign a value to the result of an update statement For example: `contractFetched <- fetch someContractId`

1.43.2.2 Reference: Templates

This page gives reference information on templates:

For the structure of a template, see *Overview: Template Structure*.

Template Name

```
template NameOfTemplate
```

This is the name of the template. It's preceded by `template` keyword. Must begin with a capital letter.
This is the highest level of nesting.
The name is used when *creating* a contract of this template (usually, from within a choice).

Template Parameters

```
with
  exampleParty : Party
  exampleParty2 : Party
  exampleParty3 : Party
  exampleParam : Text
  -- more parameters here
```

`with` keyword. The parameters are in the form of a *record type*.
Passed in when *creating* a contract from this template. These are then in scope inside the template body.
A template parameter can't have the same name as any *choice arguments* inside the template.
For all parties involved in the contract (whether they're a `signatory`, `observer`, or `controller`) you must pass them in as parameters to the contract, whether individually or as a list (`[Party]`).

Implicit Record

Whenever a template is defined, a record is implicitly defined with the same name and fields as that template. This record structure is used in Daml code to represent the data of a contract based on that template.

Note that in the general case, the existence of a local binding `b` of type `T`, where `T` is a template (and thus also a record), does not necessarily imply the existence of a contract with the same data as `b` on the ledger. You can only assume the existence of such a contract if `b` is the result of a fetch from the ledger within the same transaction.

You can create a new instance of a record of type `T` without any interaction with the ledger; in fact, this is how you construct a create command.

`this` and `self`

Within the body of a template we implicitly define a local binding `this` to represent the data of the current contract. For a template `T`, this binding is of type `T`, i.e. the implicit record defined by the template.

Within choices, you can additionally use the binding `self` to refer to the contract ID of the current contract (the one on which the choice is being executed). For a contract of template `T`, the `self` binding is of type `ContractId T`.

Template-local Definitions

```
  where
    let
      allParties = [exampleParty, exampleParty2, exampleParty3]
```

`let` keyword. Starts a block and is followed by any number of definitions, just like any other `let` block.
Template parameters as well as `this` are in scope, but `self` is not.
Definitions from the `let` block can be used anywhere else in the template's `where` block.

Warning: Some uses of the `this` keyword in template-local definitions can cause an infinite loop of evaluation as a circular dependency arises. Hence any usage of:

- `key this`
- `observer this`
- `signatory this`
- interface methods applied to `this`

in a template-local let definition will result in an error generated by the infinite loop.

Signatory Parties

```
    signatory exampleParty
```

`signatory` keyword. After `where`. Followed by at least one `Party`.
Signatories are the parties (see the `Party` type) who must consent to the creation of this contract. They are the parties who would be put into an *obligable position* when this contract is created.
Daml won't let you put someone into an obligable position without their consent. So if the contract will cause obligations for a party, they *must* be a signatory. **If they haven't authorized it, you won't be able to create the contract.** In this situation, you may see errors like:
`NameOfTemplate requires authorizers Party1,Party2,Party, but only Party1 were given.`
When a signatory consents to the contract creation, this means they also authorize the consequences of *choices* that can be exercised on this contract.
The contract is visible to all signatories (as well as the other stakeholders of the contract). That is, the compiler automatically adds signatories as observers.
Each template **must** have at least one signatory. A signatory declaration consists of the *signatory* keyword followed by a comma-separated list of one or more expressions, each expression denoting a `Party` or collection thereof.

Observers

```
observer exampleParty2
```

`observer` keyword. After `where`. Followed by at least one `Party`.
Observers are additional stakeholders, so the contract is visible to these parties (see the `Party` type).
Optional. You can have many, either as a comma-separated list or reusing the keyword. You could pass in a list (of type `[Party]`).
Use when a party needs visibility on a contract, or be informed or contract events, but is not a *signatory* or *controller*.
If you start your choice with `choice` rather than `controller` (see *Choices* below), you must make sure to add any potential controller as an observer. Otherwise, they will not be able to exercise the choice, because they won't be able to see the contract.

Choices

```
-- option 1 for specifying choices: choice name first
choice NameOfChoice1
  : ()  -- replace () with the actual return type
  with
    exampleParameter : Text -- parameters here
  controller exampleParty
  do
    return () -- replace this line with the choice body

-- option 2 for specifying choices (deprecated syntax): controller first
controller exampleParty can
  NameOfChoice2
    : () -- replace () with the actual return type
    with
      exampleParameter : Text -- parameters here
    do
      return () -- replace this line with the choice body
  nonconsuming NameOfChoice3
    : ()  -- replace () with the actual return type
    with
      exampleParameter : Text -- parameters here
    do
      return () -- replace this line with the choice body
```

A right that the contract gives the controlling party. Can be *exercised*.
This is essentially where all the logic of the template goes.
By default, choices are *consuming*: that is, exercising the choice archives the contract, so no further choices can be exercised on it. You can make a choice non-consuming using the `nonconsuming` keyword.
There are two ways of specifying a choice: start with the `choice` keyword or start with the `controller` keyword.
Starting with `choice` lets you pass in a `Party` to use as a controller. But you must make sure to add that party as an `observer`.
See *Reference: Choices* for full reference information.

Serializable Types

Every parameter to a template, choice argument, and choice result must have a *serializable type*. This does not merely mean convertible to bytes ; it has a specific meaning in Daml. The serializability rule serves three purposes:

1. Offer a stable means to store ledger values permanently.
2. Provide a sensible encoding of them over *The Ledger API*.
3. Provide sensible *types* that directly match their Daml counterparts in languages like Java for language codegen.

For example, certain kinds of type parameters Daml offers are compatible with (1) and (2), but have no proper counterpart in (3), so they are disallowed. Similarly, function types have sensible Java counterparts, satisfying (3), but no reliable way to store or share them via the API, thus failing (1) and (2).

The following types are *not serializable*, and thus may not be used in templates.

> Function types.
> Record types with any non-serializable field.
> Variant types with any non-serializable value case.
> Variant and enum types with no constructors.
> References to a parameterized data type with any non-serializable type argument. This applies whether or not the data type definition uses the type parameter.
> Defined data types with any type parameter of kind `Nat`, or any kind other than `*`. This means higher-kinded types, and types that take a parameter just to pass to `Numeric`, are not serializable.

Agreements

```
agreement
  -- text representing the contract
  ""
```

> `agreement` keyword, followed by text.
> Represents what the contract means in text. They're usually the boundary between on-ledger and off-ledger rights and obligations.
> Usually, they look like `agreement tx`, where `tx` is of type `Text`.
> You can use the built-in operator `show` to convert party names to a string, and concatenate with `<>`.

Preconditions

```
ensure
  True -- a boolean condition goes here
```

> `ensure` keyword, followed by a boolean condition.
> Used on contract creation. `ensure` limits the values on parameters that can be passed to the contract: the contract can only be created if the boolean condition is true.

Contract Keys and Maintainers

```
key (exampleParty, exampleParam) : (Party, Text)
maintainer (exampleFunction key)
```

`key` and `maintainer` keywords.
This feature lets you specify a key that you can use to uniquely identify this contract as an instance of this template.
If you specify a `key`, you must also specify a `maintainer`. This is a `Party` that will ensure the uniqueness of all the keys it is aware of.
Because of this, the `key` must include the `maintainer` `Party` or parties (for example, as part of a tuple or record), and the `maintainer` must be a signatory.
For a full explanation, see *Reference: Contract Keys*.

Interface Instances

```
interface instance MyInterface for NameOfTemplate where
  view = MyInterfaceViewType "NameOfTemplate" 100
  method1 = field1
  method2 = field2
  method3 False _ _ = 0
  method3 True x y
    | x > 0 = x + y
    | otherwise = y
```

Used to make a template an instance of an existing interface.
The clause must start with the keywords `interface instance`, followed by the name of the interface, then the keyword `for` and the name of the template (which must match the enclosing declaration), and finally the keyword `where`, which introduces a block where **all** the methods of the interface must be implemented.
See *Reference: Interfaces* for full reference information on interfaces, or section *Interface Instances* for interface instances specifically.

1.43.2.3 Reference: Choices

This page gives reference information on choices. For information on the high-level structure of a choice, see *Overview: Template Structure*.

`choice` First or `controller` First

There are two ways you can start a choice:

start with the `choice` keyword
start with the `controller` keyword

Warning: `controller` first syntax is deprecated since Daml 2.0 and will be removed in a future version. For more information, see *Deprecation of controller first syntax*.

```
    -- option 1 for specifying choices: choice name first
    choice NameOfChoice
      : () -- replace () with the actual return type
      with
        party : Party -- parameters here
      controller party
      do
        return () -- replace this line with the choice body

    -- option 2 for specifying choices (deprecated syntax): controller first
    controller exampleParty can
      NameOfAnotherChoice
        : () -- replace () with the actual return type
        with
          party : Party -- parameters here
        do
          return () -- replace the line with the choice body
```

The main difference is that starting with `choice` means that you can pass in a `Party` to use as a controller. If you do this, you **must** make sure that you add that party as an `observer`, otherwise they won't be able to see the contract (and therefore won't be able to exercise the choice).

In contrast, if you start with `controller`, the `controller` is automatically added as an observer when you compile your Daml files.

A secondary difference is that starting with `choice` allows *choice observers* to be attached to the choice using the `observer` keyword. The choice observers are a list of parties that, in addition to the stakeholders, will see all consequences of the action.

```
    -- choice observers may be specified if option 1 is used
    choice NameOfChoiceWithObserver
      : () -- replace () with the actual return type
      with
        party : Party -- parameters here
      observer party -- optional specification of choice observers (only
↪available in Daml-LF >=1.11)
      controller exampleParty
      do
        return () -- replace this line with the choice body
```

Choice Name

Listing 60: Option 1 for specifying choices: choice name first

```
    choice ExampleChoice1
      : () -- replace () with the actual return type
```

Listing 61: Option 2 for specifying choices (deprecated syntax): controller first

```
    ExampleChoice2
      : () -- replace () with the actual return type
```

The name of the choice. Must begin with a capital letter.
If you're using choice-first, preface with `choice`. Otherwise, this isn't needed.
Must be unique in the module. Different templates defined in the same module cannot share a choice name.
If you're using controller-first, you can have multiple choices after one `can`, for tidiness. However, note that this syntax is deprecated and will be removed in a future version of Daml.

Controllers

Listing 62: Option 1 for specifying choices: choice name first

```
    controller exampleParty
```

Listing 63: Option 2 for specifying choices (deprecated syntax): controller first

```
  controller exampleParty can
```

`controller` keyword
The controller is a comma-separated list of values, where each value is either a party or a collection of parties.
The conjunction of **all** the parties are required to authorize when this choice is exercised.

Contract Consumption

If no qualifier is present, choices are *consuming*: the contract is archived before the evaluation of the choice body and both the controllers and all contract stakeholders see all consequences of the action.

Preconsuming Choices

Listing 64: Option 1 for specifying choices: choice name first

```
  preconsuming choice ExampleChoice5
    : () -- replace () with the actual return type
```

Listing 65: Option 2 for specifying choices (deprecated syntax): controller first

```
    preconsuming ExampleChoice7
      : () -- replace () with the actual return type
```

`preconsuming` keyword. Optional.
Makes a choice pre-consuming: the contract is archived before the body of the exercise is executed.
The create arguments of the contract can still be used in the body of the exercise, but cannot be fetched by its contract id.
The archival behavior is analogous to the *consuming* default behavior.
Only the controllers and signatories of the contract see all consequences of the action. Other stakeholders merely see an archive action.
Can be thought as a non-consuming choice that implicitly archives the contract before anything else happens

Postconsuming Choices

Listing 66: Option 1 for specifying choices: choice name first

```
  postconsuming choice ExampleChoice6
    : () -- replace () with the actual return type
```

Listing 67: Option 2 for specifying choices (deprecated syntax): controller first

```
    postconsuming ExampleChoice8
      : () -- replace () with the actual return type
```

`postconsuming` keyword. Optional.
Makes a choice post-consuming: the contract is archived after the body of the exercise is executed.
The create arguments of the contract can still be used in the body of the exercise as well as the contract id for fetching it.
Only the controllers and signatories of the contract see all consequences of the action. Other stakeholders merely see an archive action.
Can be thought as a non-consuming choice that implicitly archives the contract after the choice has been exercised

Non-consuming Choices

Listing 68: Option 1 for specifying choices: choice name first

```
nonconsuming choice ExampleChoice3
  : () -- replace () with the actual return type
```

Listing 69: Option 2 for specifying choices (deprecated syntax): controller first

```
nonconsuming ExampleChoice4
  : () -- replace () with the actual return type
```

`nonconsuming` keyword. Optional.
Makes a choice non-consuming: that is, exercising the choice does not archive the contract.
Only the controllers and signatories of the contract see all consequences of the action.
Useful in the many situations when you want to be able to exercise a choice more than once.

Return Type

Return type is written immediately after choice name.
All choices have a return type. A contract returning nothing should be marked as returning a unit , ie `()`.
If a contract is/contracts are created in the choice body, usually you would return the contract ID(s) (which have the type `ContractId <name of template>`). This is returned when the choice is exercised, and can be used in a variety of ways.

Choice Arguments

```
with
  exampleParameter : Text
```

`with` keyword.
Choice arguments are similar in structure to *Template Parameters*: a *record type*.
A choice argument can't have the same name as any *parameter to the template* the choice is in.
Optional - only if you need extra information passed in to exercise the choice.

Choice Body

Introduced with `do`
The logic in this section is what is executed when the choice gets exercised.
The choice body contains `Update` expressions. For detail on this, see *Reference: Updates*.
By default, the last expression in the choice is returned. You can return multiple updates in tuple form or in a custom data type. To return something that isn't of type `Update`, use the `return` keyword.

Deprecation of `controller` first syntax

Since Daml 2.0, using `controller` first syntax to define a choice will result in the following warning:

```
The syntax 'controller ... can' is deprecated,
it will be removed in a future version of Daml.
Instead, use 'choice ... with ... controller' syntax.
Note that 'choice ... with ... controller' syntax does not
implicitly add the controller as an observer,
so it must be added explicitly as one (or as a signatory).
```

Migrating

Users are strongly encouraged to adapt their choices to use `choice` first syntax. This is a schema to adapt affected code:

1. For each `controller ... can` block,
    1. Note the parties between the `controller` and `can` keywords; these are the block controllers.
    2. Ensure that all the block controllers are signatories or observers of the template. If any controller is neither a signatory nor observer of the template, add it as an observer.
    3. For each choice in the block,
        1. Prefix the choice name with the `choice` keyword, but keep any consumption qualifiers before `choice`.
        2. Add a `controller` clause with the block controllers before the body of the choice (the `do` block) .
    4. Remove the `controller ... can` block header and adjust indentation as necessary.

Turning off the warning

This warning is controlled by the warning flag `controller-can`, which means that it can be toggled independently of other warnings. This is especially useful for gradually migrating code that used this syntax.

To turn off the warning within a Daml file, add the following line at the top of the file:

```
{-# OPTIONS_GHC -Wno-controller-can #-}
```

To turn it off for an entire Daml project, add the following entry to the `build-options` field of the project's `daml.yaml` file

```
build-options:
- --ghc-option=-Wno-controller-can
```

Within a project where the warning has been turned off via the `daml.yaml` file, it can be turned back on for individual Daml files by adding the following line at the top of each file:

```
{-# OPTIONS_GHC -Wcontroller-can #-}
```

1.43.2.4 Reference: Updates

This page gives reference information on Updates. For the structure around them, see *Overview: Template Structure*.

Background

An `Update` is ledger update. There are many different kinds of these, and they're listed below. They are what can go in a *choice body*.

Binding Variables

```
boundVariable <- UpdateExpression1
```

One of the things you can do in a choice body is bind (assign) an Update expression to a variable. This works for any of the Updates below.

do

```
do
   updateExpression1
   updateExpression2
```

`do` can be used to group `Update` expressions. You can only have one update expression in a choice, so any choice beyond the very simple will use a `do` block.
Anything you can put into a choice body, you can put into a `do` block.
By default, `do` returns whatever is returned by the **last expression in the block**.
So if you want to return something else, you'll need to use `return` explicitly - see *return* for an example.

archive

```
archive ContractId
```

`archive` function.
Archives a contract already created and residing on the ledger. The contract is fetched by its unique contract identifier `ContractId <name of template>` and then exercises the *Archive* choice on it.
Returns unit.
Requires authorization from the contract controllers/signatories. Without the required authorization, the transaction fails. For more detail on authorization, see *Signatory Parties*.
All templates implicitly have an *Archive* choice that cannot be removed, which is equivalent to:

```
choice Archive : ()
  controller (signatory this)
  do return ()
```

create

```
create NameOfTemplate with exampleParameters
```

`create` function.
Creates a contract on the ledger. When a contract is committed to the ledger, it is given a unique contract identifier of type `ContractId <name of template>`.
Creating the contract returns that `ContractId`.
Use `with` to specify the template parameters.
Requires authorization from the signatories of the contract being created. This is given by being signatories of the contract from which the other contract is created, being the controller, or explicitly creating the contract itself.
If the required authorization is not given, the transaction fails. For more detail on authorization, see *Signatory Parties*.

exercise

```
exercise IdOfContract NameOfChoiceOnContract with choiceArgument1 = value1
```

`exercise` function.
Exercises the specified choice on the specified contract.
Use `with` to specify the choice parameters.
Requires authorization from the controller(s) of the choice. If the authorization is not given, the transaction fails.

exerciseByKey

```
exerciseByKey @ContractType contractKey NameOfChoiceOnContract with
↪choiceArgument1 = value1
```

`exerciseByKey` function.
Like `exercise`, but the contract is specified by *contract key*, instead of contract ID.
For details see *Reference: Contract Keys: exerciseByKey*

fetch

```
fetchedContract <- fetch IdOfContract
```

`fetch` function.
Fetches the contract with that ID. Usually used with a bound variable, as in the example above. Often used to check the details of a contract before exercising a choice on that contract. Also used when referring to some reference data.
`fetch cid` fails if `cid` is not the contract id of an active contract, and thus causes the entire transaction to abort.
The submitting party must be an observer or signatory on the contract, otherwise `fetch` fails, and similarly causes the entire transaction to abort.

fetchByKey

```
fetchedContract <- fetchByKey @ContractType contractKey
```

`fetchByKey` function.
Like `fetch`, but fetches the contract with that *contract key*, instead of the contract ID.
For details see *Reference: Contract Keys: fetchByKey*.

visibleByKey

```
isVisible <- visibleByKey @ContractType contractKey
```

`visibleByKey` function.
Use this to check whether a contract with the given *contract key* exists.
For details see *Reference: Contract Keys: visibleByKey*

lookupByKey

```
fetchedContractId <- lookupByKey @ContractType contractKey
```

`lookupByKey` function.
Use this to confirm that a contract with the given *contract key* exists.
For details see *Reference: Contract Keys: lookupByKey*

abort

```
abort errorMessage
```

`abort` function.
Fails the transaction - nothing in it will be committed to the ledger.
`errorMessage` is of type `Text`. Use the error message to provide more context to an external system (e.g., it gets displayed in Daml Studio script results).
You could use `assert False` as an alternative.

assert

```
assert (condition == True)
```

`assert` keyword.
Fails the transaction if the condition is false. So the choice can only be exercised if the boolean expression evaluates to `True`.
Often used to restrict the arguments that can be supplied to a contract choice.

Here's an example of using `assert` to prevent a choice being exercised if the `Party` passed as a parameter is on a blacklist:

```
    choice Transfer : ContractId RestrictedPayout
      with newReceiver : Party
      controller receiver
      do
        assert (newReceiver /= blacklisted)
        create RestrictedPayout with receiver = newReceiver; giver; blacklisted;
↪qty
```

getTime

```
currentTime <- getTime
```

`getTime` keyword.
Gets the ledger time. (You will usually want to immediately bind it to a variable in order to be able to access the value.)
Used to restrict when a choice can be made. For example, with an `assert` that the time is later than a certain time.

Here's an example of a choice that uses a check on the current time:

```
    choice Complete : ()
      controller party
      do
        -- bind the ledger effective time to the tchoose variable using getTime
        tchoose <- getTime
        -- assert that tchoose is no earlier than the begin time
        assert (begin <= tchoose && tchoose < addRelTime begin period)
```

return

```
return ()
```

`return` keyword.
Used to return a value from `do` block that is not of type `Update`.

Here's an example where two contracts are created in a choice and both their ids are returned as a tuple:

```
do
  firstContract <- create SomeContractTemplate with arg1; arg2
  secondContract <- create SomeContractTemplate with arg1; arg2
  return (firstContract, secondContract)
```

let

See the documentation on *Let*.

Let looks similar to binding variables, but it's very different! This code example shows how:

```
do
  -- defines a function, createdContract, taking a single argument that when
  -- called _will_ create the new contract using argument for issuer and owner
  let createContract x = create NameOfContract with issuer = x; owner = x

  createContract party1
  createContract party2
```

this

`this` lets you refer to the current contract from within the choice body. This refers to the contract, *not* the contract ID.

It's useful, for example, if you want to pass the current contract to a helper function outside the template.

1.43.2.5 Reference: Data Types

This page gives reference information on Daml's data types.

Built-in Types

Table of built-in primitive types

Type	For	Example	Notes
`Int`	integers	`1`, `1000000`, `1_000_000`	`Int` values are signed 64-bit integers which represent numbers between `-9,223,372,036,854,775,808` and `9,223,372,036,854,775,807` inclusive. Arithmetic operations raise an error on overflows and division by `0`. To make long numbers more readable you can optionally add underscores.
`Decimal`	short for `Numeric 10`	`1.0`	`Decimal` values are rational numbers with precision 38 and scale 10.
`Numeric n`	fixed point decimal numbers	`1.0`	`Numeric n` values are rational numbers with `38` total digits. The scale parameter `n` controls the number of digits after the decimal point, so for example, `Numeric 10` values have 28 digits before the decimal point and 10 digits after it, and `Numeric 20` values have 18 digits before the decimal point and 20 digits after it. The value of `n` must be between `0` and `37` inclusive.
`BigNumeric`	large fixed point decimal numbers	`1.0`	`BigNumeric` values are rational numbers with up to `2^16` decimal digits. They can have up to `2^15` digits before the decimal point, and up to `2^15` digits after the decimal point.
`Text`	strings	`"hello"`	`Text` values are strings of characters enclosed by double quotes.
`Bool`	boolean values	`True`, `False`	
`Party`	unicode string representing a party	`alice <- getParty "Alice"`	Every *party* in a Daml system has a unique identifier of type `Party`. To create a value of type `Party`, use binding on the result of calling `getParty`. The party text can only contain alphanumeric characters, `-`, `_` and spaces.
`Date`	models dates	`date 2007 Apr 5`	Permissible dates range from `0001-01-01` to `9999-12-31` (using a year-month-day format). To create a value of type `Date`, use the function `date` (to get this function, import `DA.Date`).
`Time`	models absolute time (UTC)	`time (date 2007 Apr 5) 14 30 05`	`Time` values have microsecond precision with allowed range from `0001-01-01` to `9999-12-31` (using a year-month-day format). To create a value of type `Time`, use a `Date` and the function `time` (to get this function, import `DA.Time`).
`RelTime`	models differences between time values	`seconds 1`, `seconds (-2)`	`RelTime` values have microsecond precision with allowed range from *-9,223,372,036,854,775,808ms* to *9,223,372,036,854,775,807ms* There are no literals for `RelTime`. Instead they are created using one of `days`, `hours`, `minutes`, `seconds`, `milliseconds` and `microseconds` (to get these functions, import `DA.Time`).

Escaping Characters

`Text` literals support backslash escapes to include their delimiter (`\"`) and a backslash itself (`\\`).

Time

Definition of time on the ledger is a property of the execution environment. Daml assumes there is a shared understanding of what time is among the stakeholders of contracts.

Lists

`[a]` is the built-in data type for a list of elements of type `a`. The empty list is denoted by `[]` and `[1, 3, 2]` is an example of a list of type `[Int]`.

You can also construct lists using `[]` (the empty list) and `::` (which is an operator that appends an element to the front of a list). For example:

```
twoEquivalentListConstructions =
  script do
    assert ( [1, 2, 3] == 1 :: 2 :: 3 :: [] )
```

Sum a List

To sum a list, use a *fold* (because there are no loops in Daml). See *Fold* for details.

Records and Record Types

You declare a new record type using the `data` and `with` keyword:

```
data MyRecord = MyRecord
  with
    label1 : type1
    label2 : type2
    ...
    labelN : typeN
  deriving (Eq, Show)
```

where:

`label1`, `label2`, , `labelN` are *labels*, which must be unique in the record type
`type1`, `type2`, , `typeN` are the types of the fields

There's an alternative way to write record types:

```
data MyRecord = MyRecord { label1 : type1; label2 : type2; ...; labelN : typeN }
  deriving (Eq, Show)
```

The format using `with` and the format using `{ }` are exactly the same syntactically. The main difference is that when you use `with`, you can use newlines and proper indentation to avoid the delimiting semicolons.

The `deriving (Eq, Show)` ensures the data type can be compared (using ==) and displayed (using `show`). The line starting `deriving` is required for data types used in fields of a `template`.

In general, add the `deriving` unless the data type contains function types (e.g. `Int -> Int`), which cannot be compared or shown.

For example:

```
-- This is a record type with two fields, called first and second,
-- both of type `Int`
data MyRecord = MyRecord with first : Int; second : Int
  deriving (Eq, Show)

-- An example value of this type is:
newRecord = MyRecord with first = 1; second = 2

-- You can also write:
newRecord = MyRecord 1 2
```

Data Constructors

You can use `data` keyword to define a new data type, for example `data Floor a = Floor a` for some type `a`.

The first `Floor` in the expression is the *type constructor*. The second `Floor` is a *data constructor* that can be used to specify values of the `Floor Int` type: for example, `Floor 0`, `Floor 1`.

In Daml, data constructors may take *at most one argument*.

An example of a data constructor with zero arguments is `data Empty = Empty {}`. The only value of the `Empty` type is `Empty`.

Note: In `data Confusing = Int`, the `Int` is a data constructor with no arguments. It has nothing to do with the built-in `Int` type.

Access Record Fields

To access the fields of a record type, use dot notation. For example:

```
-- Access the value of the field `first`
val.first

-- Access the value of the field `second`
val.second
```

Update Record Fields

You can also use the `with` keyword to create a new record on the basis of an existing replacing select fields.

For example:

```
myRecord = MyRecord with first = 1; second = 2

myRecord2 = myRecord with second = 5
```

produces the new record value `MyRecord with first = 1; second = 5`.

If you have a variable with the same name as the label, Daml lets you use this without assigning it to make things look nicer:

```
-- if you have a variable called `second` equal to 5
second = 5

-- you could construct the same value as before with
myRecord2 = myRecord with second = second

-- or with
myRecord3 = MyRecord with first = 1; second = second

-- but Daml has a nicer way of putting this:
myRecord4 = MyRecord with first = 1; second

-- or even
myRecord5 = r with second
```

Note: The `with` keyword binds more strongly than function application. So for a function, say `return`, either write `return IntegerCoordinate with first = 1; second = 5` or `return (IntegerCoordinate {first = 1; second = 5})`, where the latter expression is enclosed in parentheses.

Parameterized Data Types

Daml supports parameterized data types.

For example, to express a more general type for 2D coordinates:

```
-- Here, a and b are type parameters.
-- The Coordinate after the data keyword is a type constructor.
data Coordinate a b = Coordinate with first : a; second : b
```

An example of a type that can be constructed with `Coordinate` is `Coordinate Int Int`.

Type Synonyms

To declare a synonym for a type, use the `type` keyword.

For example:

```
type IntegerTuple = (Int, Int)
```

This makes `IntegerTuple` and `(Int, Int)` synonyms: they have the same type and can be used interchangeably.

You can use the `type` keyword for any type, including *Built-in Types*.

Function Types

A function's type includes its parameter and result types. A function `foo` with two parameters has type `ParamType1 -> ParamType2 -> ReturnType`.

Note that this can be treated as any other type. You could for instance give it a synonym using `type FooType = ParamType1 -> ParamType2 -> ReturnType`.

Algebraic Data Types

An algebraic data type is a composite type: a type formed by a combination of other types. The enumeration data type is an example. This section introduces more powerful algebraic data types.

Product Types

The following data constructor is not valid in Daml: `data AlternativeCoordinate a b = AlternativeCoordinate a b`. This is because data constructors can only have one argument.

To get around this, wrap the values in a *record*: `data Coordinate a b = Coordinate {first: a; second: b}`.

These kinds of types are called *product* types.

A way of thinking about this is that the `Coordinate Int Int` type has a first and second dimension (that is, a 2D product space). By adding an extra type to the record, you get a third dimension, and so on.

Sum Types

Sum types capture the notion of being of one kind or another.

An example is the built-in data type `Bool`. This is defined by `data Bool = True | False deriving (Eq,Show)`, where `True` and `False` are data constructors with zero arguments . This means that a `Bool` value is either `True` or `False` and cannot be instantiated with any other value.

Please note that all types which you intend to use as template or choice arguments need to derive at least from *(Eq, Show)*.

A very useful sum type is `data Optional a = None | Some a deriving (Eq,Show)`. It is part of the *Daml standard library*.

`Optional` captures the concept of a box, which can be empty or contain a value of type `a`.

`Optional` is a sum type constructor taking a type `a` as parameter. It produces the sum type defined by the data constructors `None` and `Some`.

The `Some` data constructor takes one argument, and it expects a value of type `a` as a parameter.

Pattern Matching

You can match a value to a specific pattern using the `case` keyword.

The pattern is expressed with data constructors. For example, the `Optional Int` sum type:

```
import Daml.Script
import DA.Assert

optionalIntegerToText (x : Optional Int) : Text =
  case x of
    None -> "Box is empty"
    Some val -> "The content of the box is " <> show val

optionalIntegerToTextTest =
  script do
```

In the `optionalIntegerToText` function, the `case` construct first tries to match the `x` argument against the `None` data constructor, and in case of a match, the "`Box is empty`" text is returned. In case of no match, a match is attempted for `x` against the next pattern in the list, i.e., with the `Some` data constructor. In case of a match, the content of the value attached to the `Some` label is bound to the `val` variable, which is then used in the corresponding output text string.

Note that all patterns in the case construct need to be *complete*, i.e., for each `x` there must be at least one pattern that matches. The patterns are tested from top to bottom, and the expression for the first pattern that matches will be executed. Note that _ can be used as a catch-all pattern.

You could also case distinguish a `Bool` variable using the `True` and `False` data constructors and achieve the same behavior as an if-then-else expression.

As an example, the following is an expression for a `Text`:

```
tmp =
  let
    l = [1, 2, 3]
  in case l of
```

Notice the use of nested pattern matching above.

Note: An underscore was used in place of a variable name. The reason for this is that *Daml Studio* produces a warning for all variables that are not being used. This is useful in detecting unused variables. You can suppress the warning by naming the variable with an initial underscore.

1.43.2.6 Reference: Built-in Functions

This page gives reference information on built-in functions for working with a variety of common concepts.

Work with Time

Daml has these built-in functions for working with time:

`datetime`: creates a `Time` given year, month, day, hours, minutes, and seconds as argument.
`subTime`: subtracts one time from another. Returns the `RelTime` difference between `time1` and `time2`.
`addRelTime`: add times. Takes a `Time` and `RelTime` and adds the `RelTime` to the `Time`.
`days`, `hours`, `minutes`, `seconds`: constructs a `RelTime` of the specified length.
`pass`: (in *Daml Script tests* only) use `pass : RelTime -> Script Time` to advance the ledger time by the argument amount. Returns the new time.

Work with Numbers

Daml has these built-in functions for working with numbers:

`round`: rounds a `Decimal` number to `Int`.
`round d` is the *nearest* `Int` to `d`. Tie-breaks are resolved by rounding away from zero, for example:

```
round 2.5 == 3    round (-2.5) == -3
round 3.4 == 3    round (-3.7) == -4
```

`truncate`: converts a `Decimal` number to `Int`, truncating the value towards zero, for example:

```
truncate 2.2 == 2    truncate (-2.2) == -2
truncate 4.9 == 4    v (-4.9) == -4
```

`intToDecimal`: converts an `Int` to `Decimal`.

The set of numbers expressed by `Decimal` is not closed under division as the result may require more than 10 decimal places to represent. For example, `1.0 / 3.0 == 0.3333...` is a rational number, but not a `Decimal`.

Work with Text

Daml has these built-in functions for working with text:

`<>` operator: concatenates two `Text` values.
`show` converts a value of the primitive types (`Bool`, `Int`, `Decimal`, `Party`, `Time`, `RelTime`) to a `Text`.

To escape text in Daml strings, use `\`:

Character	How to escape it
\	\\
"	\"
'	\'
Newline	\n
Tab	\t
Carriage return	\r
Unicode (using ! as an example)	Decimal code: \33 Octal code: \o41 Hexadecimal code: \x21

Work with Lists

Daml has these built-in functions for working with lists:

`foldl` and `foldr`: see *Fold* below.

Fold

A *fold* takes:

a binary operator
a first *accumulator* value
a list of values

The elements of the list are processed one-by-one (from the left in a `foldl`, or from the right in a `foldr`).

Note: We'd usually recommend using `foldl`, as `foldr` is usually slower. This is because it needs to traverse the whole list before starting to discharge its elements.

Processing goes like this:

1. The binary operator is applied to the first accumulator value and the first element in the list. This produces a second accumulator value.
2. The binary operator is applied to the *second* accumulator value and the second element in the list. This produces a third accumulator value.
3. This continues until there are no more elements in the list. Then, the last accumulator value is returned.

As an example, to sum up a list of integers in Daml:

```
sumList =
  script do
    assert (foldl (+) 0 [1, 2, 3] == 6)
```

1.43.2.7 Reference: Expressions

This page gives reference information for Daml expressions that are not *updates*.

Definitions

Use assignment to bind values or functions at the top level of a Daml file or in a contract template body.

Values

For example:

```
pi = 3.1415926535
```

The fact that `pi` has type `Decimal` is inferred from the value. To explicitly annotate the type, mention it after a colon following the variable name:

```
pi : Decimal = 3.1415926535
```

Functions

You can define functions. Here's an example: a function for computing the surface area of a tube:

```
tubeSurfaceArea : Decimal -> Decimal -> Decimal
tubeSurfaceArea r h  =
  2.0 * pi * r * h
```

Here you see:

the name of the function
the function's type signature `Decimal -> Decimal -> Decimal`
This means it takes two Decimals and returns another Decimal.
the definition `= 2.0 * pi * r * h` (which uses the previously defined `pi`)

Arithmetic Operators

Operator	Works for
`+`	`Int`, `Decimal`, `RelTime`
`-`	`Int`, `Decimal`, `RelTime`
`*`	`Int`, `Decimal`
`/` (integer division)	`Int`
`%` (integer remainder operation)	`Int`
`^` (integer exponentiation)	`Int`

The result of the modulo operation has the same sign as the dividend:

`7 / 3` and `(-7) / (-3)` evaluate to `2`
`(-7) / 3` and `7 / (-3)` evaluate to `-2`
`7 % 3` and `7 % (-3)` evaluate to `1`
`(-7) % 3` and `(-7) % (-3)` evaluate to `-1`

To write infix expressions in prefix form, wrap the operators in parentheses. For example, `(+) 1 2` is another way of writing `1 + 2`.

Comparison Operators

Operator	Works for
<, <=, >, >=	`Bool`, `Text`, `Int`, `Decimal`, `Party`, `Time`
==, /=	`Bool`, `Text`, `Int`, `Decimal`, `Party`, `Time`, and *identifiers of contracts* stemming from the same contract template

Logical Operators

The logical operators in Daml are:

`not` for negation, e.g., `not True == False`
`&&` for conjunction, where `a && b == and a b`
`||` for disjunction, where `a || b == or a b`

for `Bool` variables `a` and `b`.

If-then-else

You can use conditional *if-then-else* expressions, for example:

```
if owner == scroogeMcDuck then "sell" else "buy"
```

Let

To bind values or functions to be in scope beneath the expression, use the block keyword `let`:

```
doubled =
  -- let binds values or functions to be in scope beneath the expression
  let
    double (x : Int) = 2 * x
    up = 5
  in double up
```

You can use `let` inside `do` blocks:

```
blah = script
  do
    let
      x = 1
```

(continues on next page)

(continued from previous page)

```
      y = 2
      -- x and y are in scope for all subsequent expressions of the do block,
      -- so can be used in expression1 and expression2.
    expression1
    expression2
```

Lastly, a `template` may contain a single `let` block.

```
template Iou
  with
    issuer : Party
    owner  : Party
  where
    signatory issuer

    let updateOwner o = create this with owner = o
        updateAmount a = create this with owner = a

    -- Expressions bound in a template let block can be referenced
    -- from any and all of the signatory, consuming, ensure and
    -- agreement expressions and from within any choice do blocks.

    choice Transfer : ContractId Iou
      with newOwner : Party
      controller owner
      do
        updateOwner newOwner
```

1.43.2.8 Reference: Functions

This page gives reference information on functions in Daml.

Daml is a functional language. It lets you apply functions partially and also have functions that take other functions as arguments. This page discusses these *higher-order functions*.

Defining Functions

In *Reference: Expressions*, the `tubeSurfaceArea` function was defined as:

```
tubeSurfaceArea : Decimal -> Decimal -> Decimal
tubeSurfaceArea r h  =
  2.0 * pi * r * h
```

You can define this function equivalently using lambdas, involving `\`, a sequence of parameters, and an arrow `->` as:

```
tubeSurfaceArea : BinaryDecimalFunction =
  \ (r : Decimal) (h : Decimal) -> 2.0 * pi * r * h
```

Partial Application

The type of the `tubeSurfaceArea` function described previously, is `Decimal -> Decimal -> Decimal`. An equivalent, but more instructive, way to read its type is: `Decimal -> (Decimal -> Decimal)`: saying that `tubeSurfaceArea` is a function that takes *one* argument and returns another function.

So `tubeSurfaceArea` expects one argument of type `Decimal` and returns a function of type `Decimal -> Decimal`. In other words, this function returns another function. *Only the last application of an argument yields a non-function.*

This is called *currying*: currying is the process of converting a function of multiple arguments to a function that takes just a single argument and returns another function. In Daml, all functions are curried.

This doesn't affect things that much. If you use functions in the classical way (by applying them to all parameters) then there is no difference.

If you only apply a few arguments to the function, this is called *partial application*. The result is a function with partially defined arguments. For example:

```
multiplyThreeNumbers : Int -> Int -> Int -> Int
multiplyThreeNumbers xx yy zz =
  xx * yy * zz

multiplyTwoNumbersWith7 = multiplyThreeNumbers 7

multiplyWith21 = multiplyTwoNumbersWith7 3

multiplyWith18 = multiplyThreeNumbers 3 6
```

You could also define equivalent lambda functions:

```
multiplyWith18_v2 : Int -> Int
multiplyWith18_v2 xx =
  multiplyThreeNumbers 3 6 xx
```

Functions are Values

The function type can be explicitly added to the `tubeSurfaceArea` function (when it is written with the lambda notation):

```
-- Type synonym for Decimal -> Decimal -> Decimal
type BinaryDecimalFunction = Decimal -> Decimal -> Decimal

pi : Decimal = 3.1415926535

tubeSurfaceArea : BinaryDecimalFunction =
  \ (r : Decimal) (h : Decimal) -> 2.0 * pi * r * h
```

Note that `tubeSurfaceArea : BinaryDecimalFunction = ...` follows the same pattern as when binding values, e.g., `pi : Decimal = 3.14159265359`.

Functions have types, just like values. Which means they can be used just like normal variables. In fact, in Daml, functions are values.

This means a function can take another function as an argument. For example, define a function `applyFilter: (Int -> Int -> Bool) -> Int -> Int -> Bool` which applies the first argument, a higher-order function, to the second and the third arguments to yield the result.

```
applyFilter (filter : Int -> Int -> Bool)
    (x : Int)
    (y : Int) = filter x y

compute = script do
    applyFilter (<) 3 2 === False
    applyFilter (/=) 3 2 === True

    round (2.5 : Decimal) === 3
    round (3.5 : Decimal) === 4

    explode "me" === ["m", "e"]

    applyFilter (\a b -> a /= b) 3 2 === True
```

The *Fold* section looks into two useful built-in functions, `foldl` and `foldr`, that also take a function as an argument.

Note: Daml does not allow functions as parameters of contract templates and contract choices. However, a follow up of a choice can use built-in functions, defined at the top level or in the contract template body.

Generic Functions

A function is *parametrically polymorphic* if it behaves uniformly for all types, in at least one of its type parameters. For example, you can define function composition as follows:

```
compose (f : b -> c) (g : a -> b) (x : a) : c = f (g x)
```

where `a`, `b`, and `c` are any data types. Both `compose ((+) 4) ((*) 2) 3 == 10` and `compose not ((&&) True) False` evaluate to `True`. Note that `((+) 4)` has type `Int -> Int`, whereas `not` has type `Bool -> Bool`.

You can find many other generic functions including this one in the *Daml standard library*.

Note: Daml currently does not support generic functions for a specific set of types, such as `Int` and `Decimal` numbers. For example, `sum (x: a) (y: a) = x + y` is undefined when `a` equals the type `Party`. *Bounded polymorphism* might be added to Daml in a later version.

1.43.2.9 Reference: Daml File Structure

This page gives reference information on the structure of Daml files outside of *templates*.

File Structure

This file's module name (`module NameOfThisFile where`).
Part of a hierarchical module system to facilitate code reuse. Must be the same as the Daml file name, without the file extension.
For a file with path `./Scenarios/Demo.daml`, use `module Scenarios.Demo where`.

Imports

You can import other modules (`import OtherModuleName`), including qualified imports (`import qualified AndYetOtherModuleName`, `import qualified AndYetOtherModuleName as Signifier`). Can't have circular import references.
To import the `Prelude` module of `./Prelude.daml`, use `import Prelude`.
To import a module of `./Scenarios/Demo.daml`, use `import Scenarios.Demo`.
If you leave out `qualified`, and a module alias is specified, top-level declarations of the imported module are imported into the module's namespace as well as the namespace specified by the given alias.

Libraries

A Daml library is a collection of related Daml modules.

Define a Daml library using a `LibraryModules.daml` file: a normal Daml file that imports the root modules of the library. The library consists of the `LibraryModules.daml` file and all its dependencies, found by recursively following the imports of each module.

Errors are reported in Daml Studio on a per-library basis. This means that breaking changes on shared Daml modules are displayed even when the files are not explicitly open.

Comments

Use `--` for a single line comment. Use `{-` and `-}` for a comment extending over multiple lines.

Contract Identifiers

When an instance of a template (that is, a contract) is added to the ledger, it's assigned a unique identifier, of type `ContractId <name of template>`.

The runtime representation of these identifiers depends on the execution environment: a contract identifier from the Sandbox may look different to ones on other Daml Ledgers.

You can use `==` and `/=` on contract identifiers of the same type.

1.43.2.10 Reference: Daml Packages

This page gives reference information on Daml package dependencies.

Building Daml Archives

When a Daml project is compiled, the compiler produces a *Daml archive*. These are platform-independent packages of compiled Daml code that can be uploaded to a Daml ledger or imported in other Daml projects.

Daml archives have a `.dar` file ending. By default, when you run `daml build`, it will generate the `.dar` file in the `.daml/dist` folder in the project root folder. For example, running `daml build` in project `foo` with project version `0.0.1` will result in a Daml archive `.daml/dist/foo-0.0.1.dar`.

You can specify a different path for the Daml archive by using the `-o` flag:

```
 daml build -o foo.dar

The rest of this page will focus on how to import a Daml package in other Daml
 ↪projects.
```

Inspecting DARs

To inspect a DAR and get information about the packages inside it, you can use the `daml damlc inspect-dar` command. This is often useful to find the package id of the project you just built.

You can run `daml damlc inspect-dar /path/to/your.dar` to get a human-readable listing of the files inside it and a list of packages and their package ids. Here is a (shortened) example output:

```
$ daml damlc inspect-dar .daml/dist/create-daml-app-0.1.0.dar
DAR archive contains the following files:

create-daml-app-0.1.0-
 ↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/create-daml-
 ↪app-0.1.0-29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d.dalf
create-daml-app-0.1.0-
 ↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/daml-prim-
 ↪75b070729b1fbd37a618493652121b0d6f5983b787e35179e52d048db70e9f15.dalf
create-daml-app-0.1.0-
 ↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/daml-stdlib-0.
 ↪0.0-a535cbc3657b8df953a50aaef5a4cd224574549c83ca4377e8219aadea14f21a.dalf
create-daml-app-0.1.0-
 ↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/daml-stdlib-DA-
 ↪Internal-Template-
 ↪d14e08374fc7197d6a0de468c968ae8ba3aadbf9315476fd39071831f5923662.dalf
create-daml-app-0.1.0-
 ↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/data/create-
 ↪daml-app-0.1.0.conf
create-daml-app-0.1.0-
 ↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/User.daml
create-daml-app-0.1.0-
 ↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/User.hi
create-daml-app-0.1.0-
 ↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/User.hie
```

(continues on next page)

(continued from previous page)

```
META-INF/MANIFEST.MF

DAR archive contains the following packages:

create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d
↪"29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d"
daml-stdlib-DA-Internal-Template-
↪d14e08374fc7197d6a0de468c968ae8ba3aadbf9315476fd39071831f5923662
↪"d14e08374fc7197d6a0de468c968ae8ba3aadbf9315476fd39071831f5923662"
daml-prim-75b070729b1fbd37a618493652121b0d6f5983b787e35179e52d048db70e9f15
↪"75b070729b1fbd37a618493652121b0d6f5983b787e35179e52d048db70e9f15"
daml-stdlib-0.0.0-
↪a535cbc3657b8df953a50aaef5a4cd224574549c83ca4377e8219aadea14f21a
↪"a535cbc3657b8df953a50aaef5a4cd224574549c83ca4377e8219aadea14f21a"
```

In addition to the human-readable output, you can also get the output as JSON. This is easier to consume programmatically and it is more robust to changes across SDK versions:

```
$ daml damlc inspect-dar --json .daml/dist/create-daml-app-0.1.0.dar
{
    "packages": {
        "29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d": {
            "path": "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/create-daml-
↪app-0.1.0-29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d.dalf
↪",
            "name": "create-daml-app",
            "version": "0.1.0"
        },
        "d14e08374fc7197d6a0de468c968ae8ba3aadbf9315476fd39071831f5923662": {
            "path": "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/daml-stdlib-DA-
↪Internal-Template-
↪d14e08374fc7197d6a0de468c968ae8ba3aadbf9315476fd39071831f5923662.dalf",
            "name": null,
            "version": null
        },
        "75b070729b1fbd37a618493652121b0d6f5983b787e35179e52d048db70e9f15": {
            "path": "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/daml-prim-
↪75b070729b1fbd37a618493652121b0d6f5983b787e35179e52d048db70e9f15.dalf",
            "name": "daml-prim",
            "version": "0.0.0"
        },
        "a535cbc3657b8df953a50aaef5a4cd224574549c83ca4377e8219aadea14f21a": {
            "path": "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/daml-stdlib-0.
↪0.0-a535cbc3657b8df953a50aaef5a4cd224574549c83ca4377e8219aadea14f21a.dalf",
            "name": "daml-stdlib",
            "version": "0.0.0"
        }
    },
    "main_package_id":
↪"29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d",
```

(continues on next page)

(continued from previous page)

```
    "files": [
        "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/create-daml-
↪app-0.1.0-29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d.dalf
↪",
        "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/daml-prim-
↪75b070729b1fbd37a618493652121b0d6f5983b787e35179e52d048db70e9f15.dalf",
        "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/daml-stdlib-0.
↪0.0-a535cbc3657b8df953a50aaef5a4cd224574549c83ca4377e8219aadea14f21a.dalf",
        "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/daml-stdlib-DA-
↪Internal-Template-
↪d14e08374fc7197d6a0de468c968ae8ba3aadbf9315476fd39071831f5923662.dalf",
        "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/data/create-
↪daml-app-0.1.0.conf",
        "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/User.daml",
        "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/User.hi",
        "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/User.hie",
        "META-INF/MANIFEST.MF"
    ]
}
```

Note that `name` and `version` will be `null` for packages in Daml-LF < 1.8.

Import Daml Packages

There are two ways to import a Daml package in a project: via `dependencies`, and via `data-dependencies`. They each have certain advantages and disadvantages. To summarize:

> `dependencies` allow you to import a Daml archive as a library. The definitions in the dependency will all be made available to the importing project. However, the dependency must be compiled with the same SDK version, so this method is only suitable for breaking up large projects into smaller projects that depend on each other, or to reuse existing libraries.
> `data-dependencies` allow you to import a Daml archive (.dar) or a Daml-LF package (.dalf), including packages that have already been deployed to a ledger. These packages can be compiled with any previous SDK version. On the other hand, not all definitions can be carried over perfectly, since the Daml interface needs to be reconstructed from the binary.

The following sections will cover these two approaches in more depth.

Import a Daml package via Dependencies

A Daml project can declare a Daml archive as a dependency in the `dependencies` field of `daml.yaml`. This lets you import modules and reuse definitions from another Daml project. The main limitation of this method is that the dependency must be built for the same SDK version as the importing project.

Let's go through an example. Suppose you have an existing Daml project `foo`, located at `/home/user/foo`, and you want to use it as a dependency in a project `bar`, located at `/home/user/bar`.

To do so, you first need to generate the Daml archive of `foo`. Go into `/home/user/foo` and run `daml build -o foo.dar`. This will create the Daml archive, `/home/user/foo/foo.dar`.

Next, we will update the project config for `bar` to use the generated Daml archive as a dependency. Go into `/home/user/bar` and change the `dependencies` field in `daml.yaml` to point to the created *Daml archive*:

```
dependencies:
  - daml-prim
  - daml-stdlib
  - ../foo/foo.dar
```

The import path can also be absolute, for example, by changing the last line to:

```
- /home/user/foo/foo.dar
```

When you run `daml build` in the `bar` project, the compiler will make the definitions in `foo.dar` available for importing. For example, if `foo` exports the module `Foo`, you can import it in the usual way:

```
import Foo
```

By default, all modules of `foo` are made available when importing `foo` as a dependency. To limit which modules of `foo` get exported, you may add an `exposed-modules` field in the `daml.yaml` file for `foo`:

```
exposed-modules:
- Foo
```

Import a Daml Archive via `data-dependencies`

You can import a Daml archive (.dar) or Daml-LF package (.dalf) using `data-dependencies`. Unlike `dependencies`, this can be used when the SDK versions do not match.

For example, you can import `foo.dar` as follows:

```
dependencies:
- daml-prim
- daml-stdlib
data-dependencies:
- ../foo/foo.dar
```

When importing packages this way, the Daml compiler will try to reconstruct the original Daml interface from the compiled binaries. However, to allow `data-dependencies` to work across SDK

versions, the compiler has to abstract over some details which are not compatible across SDK versions. This means that there are some Daml features that cannot be recovered when using `data-dependencies`. In particular:

1. Export lists cannot be recovered, so imports via `data-dependencies` can access definitions that were originally hidden. This means it is up to the importing module to respect the data abstraction of the original module. Note that this is the same for all code that runs on the ledger, since the ledger does not provide special support for data abstraction.
2. If you have a `dependency` that limits the modules that can be accessed via `exposed-modules`, you can get an error if you also have a `data-dependency` that references something from the hidden modules (even if it is only reexported). Since `exposed-modules` are not available on the ledger in general, we recommend to not make use of them and instead rely on naming conventions (e.g., suffix module names with `.Internal`) to make it clear which modules are part of the public API.
3. Prior to Daml-LF version 1.8, typeclasses could not be reconstructed. This means if you have a package that is compiled with an older version of Daml-LF, typeclasses and typeclass instances will not be carried over via data-dependencies, and you won't be able to call functions that rely on typeclass instances. This includes the template functions, such as `create`, `signatory`, and `exercise`, as these rely on typeclass instances.
4. Starting from Daml-LF version 1.8, when possible, typeclass instances will be reconstructed by re-using the typeclass definitions from dependencies, such as the typeclasses exported in `daml-stdlib`. However, if the typeclass signature has changed, you will get an instance for a reconstructed typeclass instead, which will not interoperate with code from dependencies.

Because of their flexibility, data-dependencies are a tool that is recommended for performing Daml model upgrades. See the *upgrade documentation* for more details.

Reference Daml Packages Already On the Ledger

Daml packages that have been uploaded to a ledger can be imported as data dependencies, given you have the necessary permissions to download these packages. To import such a package, add the package name and version separated by a colon to the data-dependencies stanza as follows:

```
ledger:
  host: localhost
  port: 6865
dependencies:
- daml-prim
- daml-stdlib
data-dependencies:
- foo:1.0.0
```

If your ledger runs at the default host and port (`localhost:6865`), the ledger stanza can be omitted. This will fetch and install the package `foo-1.0.0`. A `daml.lock` file is created at the root of your project directory, pinning the resolved packages to their exact package ID:

```
dependencies:
- pkgId: 51255efad65a1751bcee749d962a135a65d12b87eb81ac961142196d8bbca535
  name: foo
  version: 1.0.0
```

The `daml.lock` file needs to be checked into version control of your project. This assures that package name/version tuples specified in your data dependencies are always resolved to the same pack-

age ID. To recreate or update your `daml.lock` file, delete it and run `daml build` again.

Handling Module Name Collisions

Sometimes you will have multiple packages with the same module name. In that case, a simple import will fail, since the compiler doesn't know which version of the module to load. Fortunately, there are a few tools you can use to approach this problem.

The first is to use package qualified imports. Supposing you have packages with different names, `foo` and `bar`, which both expose a module `X`, you can select which one you want with a package qualified import.

To get `X` from `foo`:

```
import "foo" X
```

To get `X` from `bar`:

```
import "bar" X
```

To get both, you need to rename the module as you perform the import:

```
import "foo" X as FooX
import "bar" X as BarX
```

Sometimes, package qualified imports will not help, because you are importing two packages with the same name. For example, if you're loading different versions of the same package. To handle this case, you need the `--package` build option.

Suppose you are importing packages `foo-1.0.0` and `foo-2.0.0`. Notice they have the same name `foo` but different versions. To get modules that are exposed in both packages, you will need to provide module aliases. You can do this by passing the `--package` build option. Open `daml.yaml` and add the following `build-options`:

```
build-options:
- '--package'
- 'foo-1.0.0 with (X as Foo1.X)'
- '--package'
- 'foo-2.0.0 with (X as Foo2.X)'
```

This will alias the `X` in `foo-1.0.0` as `Foo1.X`, and alias the `X` in `foo-2.0.0` as `Foo2.X`. Now you will be able to import both `X` by using the new names:

```
import qualified Foo1.X
import qualified Foo2.X
```

It is also possible to add a prefix to all modules in a package using the `module-prefixes` field in your `daml.yaml`. This is particularly useful for upgrades where you can map all modules of version `v` of your package under `V$v`. For the example above you can use the following:

```
module-prefixes:
  foo-1.0.0: Foo1
  foo-2.0.0: Foo2
```

That will allow you to import module `X` from package `foo-1.0.0` as `Foo1.X` and `X` from package `foo-2.0.0` as `Foo2`.

You can also use more complex module prefixes, e.g., `foo-1.0.0: Foo1.Bar` which will make module `X` available under `Foo1.Bar.X`.

1.43.2.11 Reference: Contract Keys

Contract keys are an optional addition to templates. They let you specify a way of uniquely identifying contracts, using the parameters to the template - similar to a primary key for a database.

Contract keys do not change and can be used to refer to a contract even when the contract id changes.

Here's an example of setting up a contract key for a bank account, to act as a bank account ID:

```
type AccountKey = (Party, Text)

template Account with
    bank : Party
    number : Text
    owner : Party
    balance : Decimal
    observers : [Party]
  where
    signatory [bank, owner]
    observer observers

    key (bank, number) : AccountKey
    maintainer key._1
```

What Can Be a Contract Key

The key can be an arbitrary serializable expression that does **not** contain contract IDs. However, it **must** include every party that you want to use as a `maintainer` (see *Specify Maintainers* below).

It's best to use simple types for your keys like `Text` or `Int`, rather than a list or more complex type.

Specify Maintainers

If you specify a contract key for a template, you must also specify a `maintainer` or maintainers, in a similar way to specifying signatories or observers. The maintainers own the key in the same way the signatories own a contract. Just like signatories of contracts prevent double spends or use of false contract data, maintainers of keys prevent double allocation or incorrect lookups. Since the key is part of the contract, the maintainers **must** be signatories of the contract. However, maintainers are computed from the `key` instead of the template arguments. In the example above, the `bank` is ultimately the maintainer of the key.

Uniqueness of keys is guaranteed per template. Since multiple templates may use the same key type, some key-related functions must be annotated using the `@ContractType` as shown in the examples below.

When you are writing Daml models, the maintainers matter since they affect authorization – much like signatories and observers. You don't need to do anything to maintain the keys. In the above example, it is guaranteed that there can only be one `Account` with a given `number` at a given `bank`.

Checking of the keys is done automatically at execution time, by the Daml execution engine: if someone tries to create a new contract that duplicates an existing contract key, the execution engine will cause that creation to fail.

Contract Lookups

The primary purpose of contract keys is to provide a stable, and possibly meaningful, identifier that can be used in Daml to fetch contracts. There are two functions to perform such lookups: *fetchByKey* and *lookupByKey*. Both types of lookup are performed at interpretation time on the submitting Participant Node, on a best-effort basis. Currently, that best-effort means lookups only return contracts if the submitting Party is a stakeholder of that contract.

In particular, the above means that if multiple commands are submitted simultaneously, all using contract lookups to find and consume a given contract, there will be contention between these commands, and at most one will succeed. For more information, see the section on Avoiding Contention.

Limiting key usage to stakeholders also means that keys cannot be used to access a divulged contract, i.e. there can be cases where *fetch* succeeds and *fetchByKey* does not. See the example at the end of this section for details.

fetchByKey

```
(fetchedContractId, fetchedContract) <- fetchByKey @ContractType contrac-
tKey
```

Use `fetchByKey` to fetch the ID and data of the contract with the specified key. It is an alternative to `fetch` and behaves the same in most ways.

It returns a tuple of the ID and the contract object (containing all its data).

Like `fetch`, `fetchByKey` needs to be authorized by at least one stakeholder.

`fetchByKey` fails and aborts the transaction if:

> The submitting Party is not a stakeholder on a contract with the given key, or
> A contract was found, but the `fetchByKey` violates the authorization rule, meaning no stakeholder authorized the `fetch`.

This means that if it fails, it doesn't guarantee that a contract with that key doesn't exist, just that the submitting Party doesn't know about it, or there are issues with authorization.

visibleByKey

```
boolean <- visibleByKey @ContractType contractKey
```

Use `visibleByKey` to check whether you can see an active contract for the given key with the current authorizations. If the contract exists and you have permission to see it, returns `True`, otherwise returns `False`.

To clarify, ignoring contention:

1. `visibleByKey` will return `True` if all of these are true: there exists a contract for the given key, the submitter is a stakeholder on that contract, and at the point of call we have the authorization of **all** of the maintainers of the key.
2. `visibleByKey` will return `False` if all of those are true: there is no contract for the given key, and at the point of call we have authorization from **all** the maintainers of the key.
3. `visibleByKey` will abort the transaction at interpretation time if, at the point of call, we are missing the authorization from any one maintainer of the key.
4. `visibleByKey` will fail at validation time (after returning `False` at interpretation time) if all of these are true: at the point of call, we have the authorization of **all** the maintainers, and a valid contract exists for the given key, but the submitter is not a stakeholder on that contract.

While it may at first seem too restrictive to require **all** maintainers to authorize the call, this is actually required in order to validate negative lookups. In the positive case, when you can see the contract, it's easy for the transaction to mention which contract it found, and therefore for validators to check that this contract does indeed exist, and is active as of the time of executing the transaction.

For the negative case, however, the transaction submitted for execution cannot say *which* contract it has not found (as, by definition, it has not found it, and it may not even exist). Still, validators have to be able to reproduce the result of not finding the contract, and therefore they need to be able to look for it, which means having the authorization to ask the maintainers about it.

lookupByKey

```
optionalContractId <- lookupByKey @ContractType contractKey
```

Use `lookupByKey` to check whether a contract with the specified key exists. If it does exist, `lookupByKey` returns the `Some contractId`, where `contractId` is the ID of the contract; otherwise, it returns `None`.

`lookupByKey` is conceptually equivalent to

```
lookupByKey : forall c k. (HasFetchByKey c k) => k -> Update (Optional
↪(ContractId c))
lookupByKey k = do
  visible <- visibleByKey @c k
  if visible then do
    (contractId, _ignoredContract) <- fetchByKey @c k
    return $ Some contractId
  else
    return None
```

Therefore, `lookupByKey` needs all the same authorizations as *visibleByKey*, for the same reasons, and fails in the same cases.

To get the data from the contract once you've confirmed it exists, you'll still need to use `fetch`.

exerciseByKey

`exerciseByKey @ContractType contractKey`

Use `exerciseByKey` to exercise a choice on a contract identified by its `key` (compared to `exercise`, which lets you exercise a contract identified by its `ContractId`). Just like `exercise`, running `exerciseByKey` requires visibility of the contract (either through divulgence, `readAs` or being a stakeholder) and authorization from the controllers of the choice.

Example

A complete example of possible success and failure scenarios of *fetchByKey* and *lookupByKey* is shown below.

```
-- Copyright (c) 2023 Digital Asset (Switzerland) GmbH and/or its affiliates. All
↪rights reserved.
-- SPDX-License-Identifier: Apache-2.0

module Keys where

import Daml.Script
import DA.Assert
import DA.Optional

template Keyed
  with
    sig : Party
    obs : Party
  where
    signatory sig
    observer obs

    key sig : Party
    maintainer key

template Divulger
  with
    divulgee : Party
    sig : Party
  where
    signatory divulgee
    observer sig

    nonconsuming choice DivulgeKeyed
      : Keyed
      with
        keyedCid : ContractId Keyed
      controller sig
      do
        fetch keyedCid

template Delegation
  with
    sig : Party
    delegees : [Party]
```

(continues on next page)

(continued from previous page)

```
  where
    signatory sig
    observer delegees

    nonconsuming choice CreateKeyed
      : ContractId Keyed
      with
        delegee : Party
        obs : Party
      controller delegee
      do
        create Keyed with sig; obs

    nonconsuming choice ArchiveKeyed
      : ()
      with
        delegee : Party
        keyedCid : ContractId Keyed
      controller delegee
      do
        archive keyedCid

    nonconsuming choice UnkeyedFetch
      : Keyed
      with
        cid : ContractId Keyed
        delegee : Party
      controller delegee
      do
        fetch cid

    nonconsuming choice VisibleKeyed
      : Bool
      with
        key : Party
        delegee : Party
      controller delegee
      do
        visibleByKey @Keyed key

    nonconsuming choice LookupKeyed
      : Optional (ContractId Keyed)
      with
        lookupKey : Party
        delegee : Party
      controller delegee
      do
        lookupByKey @Keyed lookupKey

    nonconsuming choice FetchKeyed
      : (ContractId Keyed, Keyed)
      with
        lookupKey : Party
        delegee : Party
      controller delegee
      do
```

(continues on next page)

(continued from previous page)

```
        fetchByKey @Keyed lookupKey

template Helper
  with
    p : Party
  where
    signatory p

    choice FetchByKey : (ContractId Keyed, Keyed)
      with
        keyedKey : Party
      controller p
      do fetchByKey @Keyed keyedKey

    choice VisibleByKey : Bool
      with
        keyedKey : Party
      controller p
      do visibleByKey @Keyed keyedKey

    choice LookupByKey : (Optional (ContractId Keyed))
      with
        keyedKey : Party
      controller p
      do lookupByKey @Keyed keyedKey

    choice AssertNotVisibleKeyed : ()
      with
        delegationCid : ContractId Delegation
        delegee : Party
        key : Party
      controller p
      do
        b <- exercise delegationCid VisibleKeyed with
          delegee
          key
        assert $ not b

    choice AssertLookupKeyedIsNone : ()
      with
        delegationCid : ContractId Delegation
        delegee : Party
        lookupKey : Party
      controller p
      do
        b <- exercise delegationCid LookupKeyed with
          delegee
          lookupKey
        assert $ isNone b

    choice AssertFetchKeyedEqExpected : ()
      with
        delegationCid : ContractId Delegation
        delegee : Party
        lookupKey : Party
        expectedCid : ContractId Keyed
```

(continues on next page)

(continued from previous page)

```
      controller p
      do
        (cid, keyed) <- exercise delegationCid FetchKeyed with
          delegee
          lookupKey
        cid === expectedCid


lookupTest = script do

  -- Put four parties in the four possible relationships with a `Keyed`
  sig <- allocateParty "s" -- Signatory
  obs <- allocateParty "o" -- Observer
  divulgee <- allocateParty "d" -- Divulgee
  blind <- allocateParty "b" -- Blind

  keyedCid <- submit sig do createCmd Keyed with ..
  divulgercid <- submit divulgee do createCmd Divulger with ..
  submit sig do exerciseCmd divulgercid DivulgeKeyed with ..

  -- Now the signatory and observer delegate their choices
  sigDelegationCid <- submit sig do
    createCmd Delegation with
      sig
      delegees = [obs, divulgee, blind]
  obsDelegationCid <- submit obs do
    createCmd Delegation with
      sig = obs
      delegees = [divulgee, blind]

  -- TESTING LOOKUPS AND FETCHES

  -- Maintainer can fetch
  (cid, keyed) <- submit sig do
    Helper sig `createAndExerciseCmd` FetchByKey sig
  cid === keyedCid
  -- Maintainer can see
  b <- submit sig do
    Helper sig `createAndExerciseCmd` VisibleByKey sig
  assert b
  -- Maintainer can lookup
  mcid <- submit sig do
    Helper sig `createAndExerciseCmd` LookupByKey sig
  mcid === Some keyedCid


  -- Stakeholder can fetch
  (cid, l) <- submit obs do
    Helper obs `createAndExerciseCmd` FetchByKey sig
  keyedCid === cid
  -- Stakeholder can't see without authorization
  submitMustFail obs do
    Helper obs `createAndExerciseCmd` VisibleByKey sig

  -- Stakeholder can see with authorization
  b <- submit obs do
```

(continues on next page)

(continued from previous page)

```
    exerciseCmd sigDelegationCid VisibleKeyed with
      delegee = obs
      key = sig
  assert b
  -- Stakeholder can't lookup without authorization
  submitMustFail obs do
    Helper obs `createAndExerciseCmd` LookupByKey sig
  -- Stakeholder can lookup with authorization
  mcid <- submit obs do
    exerciseCmd sigDelegationCid LookupKeyed with
      delegee = obs
      lookupKey = sig
  mcid === Some keyedCid

  -- Divulgee _can_ fetch the contract directly
  submit divulgee do
    exerciseCmd obsDelegationCid UnkeyedFetch with
        delegee = divulgee
        cid = keyedCid
  -- Divulgee can't fetch through the key
  submitMustFail divulgee do
    Helper divulgee `createAndExerciseCmd` FetchByKey sig
  -- Divulgee can't see
  submitMustFail divulgee do
    Helper divulgee `createAndExerciseCmd` VisibleByKey sig
  -- Divulgee can't see with stakeholder authority
  submitMustFail divulgee do
    exerciseCmd obsDelegationCid VisibleKeyed with
        delegee = divulgee
        key = sig
  -- Divulgee can't lookup
  submitMustFail divulgee do
    Helper divulgee `createAndExerciseCmd` LookupByKey sig
  -- Divulgee can't lookup with stakeholder authority
  submitMustFail divulgee do
    exerciseCmd obsDelegationCid LookupKeyed with
        delegee = divulgee
        lookupKey = sig
  -- Divulgee can't do positive lookup with maintainer authority.
  submitMustFail divulgee do
    Helper divulgee `createAndExerciseCmd` AssertNotVisibleKeyed with
      delegationCid = sigDelegationCid
      delegee = divulgee
      key = sig
  -- Divulgee can't do positive lookup with maintainer authority.
  -- Note that the lookup returns `None` so the assertion passes.
  -- If the assertion is changed to `isSome`, the assertion fails,
  -- which means the error message changes. The reason is that the
  -- assertion is checked at interpretation time, before the lookup
  -- is checked at validation time.
  submitMustFail divulgee do
    Helper divulgee `createAndExerciseCmd` AssertLookupKeyedIsNone with
      delegationCid = sigDelegationCid
      delegee = divulgee
      lookupKey = sig
  -- Divulgee can't fetch with stakeholder authority
```

(continues on next page)

(continued from previous page)

```
  submitMustFail divulgee do
    Helper divulgee `createAndExerciseCmd` AssertFetchKeyedEqExpected with
      delegationCid = obsDelegationCid
      delegee = divulgee
      lookupKey = sig
      expectedCid = keyedCid

  -- Blind party can't fetch
  submitMustFail blind do
    Helper blind `createAndExerciseCmd` FetchByKey sig
  -- Blind party can't see
  submitMustFail blind do
    Helper blind `createAndExerciseCmd` VisibleByKey sig
  -- Blind party can't see with stakeholder authority
  submitMustFail blind do
    exerciseCmd obsDelegationCid VisibleKeyed with
      delegee = blind
      key = sig
  -- Blind party can't see with maintainer authority
  submitMustFail blind do
    Helper blind `createAndExerciseCmd` AssertNotVisibleKeyed with
      delegationCid = sigDelegationCid
      delegee = blind
      key = sig
  -- Blind party can't lookup
  submitMustFail blind do
    Helper blind `createAndExerciseCmd` LookupByKey sig
  -- Blind party can't lookup with stakeholder authority
  submitMustFail blind do
    exerciseCmd obsDelegationCid LookupKeyed with
      delegee = blind
      lookupKey = sig
  -- Blind party can't lookup with maintainer authority.
  -- The lookup initially returns `None`, but is rejected at
  -- validation time
  submitMustFail blind do
    Helper blind `createAndExerciseCmd` AssertLookupKeyedIsNone with
      delegationCid = sigDelegationCid
      delegee = blind
      lookupKey = sig
  -- Blind party can't fetch with stakeholder authority as lookup is negative
  submitMustFail blind do
    exerciseCmd obsDelegationCid FetchKeyed with
      delegee = blind
      lookupKey = sig
  -- Blind party can see nonexistence of a contract
  submit blind do
    Helper blind `createAndExerciseCmd` AssertNotVisibleKeyed with
      delegationCid = obsDelegationCid
      delegee = blind
      key = obs
  -- Blind can do a negative lookup on a truly nonexistant contract
  submit blind do
    Helper blind `createAndExerciseCmd` AssertLookupKeyedIsNone with
      delegationCid = obsDelegationCid
      delegee = blind
```

(continues on next page)

(continued from previous page)

```
    lookupKey = obs

-- TESTING CREATES AND ARCHIVES

-- Divulgee can archive
submit divulgee do
  exerciseCmd sigDelegationCid ArchiveKeyed with
    delegee = divulgee
    keyedCid
-- Divulgee can create
keyedCid2 <- submit divulgee do
  exerciseCmd sigDelegationCid CreateKeyed with
    delegee = divulgee
    obs

-- Stakeholder can archive
submit obs do
  exerciseCmd sigDelegationCid ArchiveKeyed with
    delegee = obs
    keyedCid = keyedCid2
-- Stakeholder can create
keyedCid3 <- submit obs do
  exerciseCmd sigDelegationCid CreateKeyed with
    delegee = obs
    obs

return ()
```

1.43.2.12 Reference: Exceptions

Exceptions are a Daml feature which provides a way to handle certain errors that arise during interpretation instead of aborting the transaction, and to roll back the state changes that lead to the error.

There are two types of errors:

Builtin Errors

Exception type	Thrown on
`GeneralError`	Calls to `error` and `abort`
`ArithmeticError`	Arithmetic errors like overflows and division by zero
`PreconditionFailed`	`ensure` statements that return `False`
`AssertionFailed`	Failed `assert` calls (or other functions from `DA.Assert`)

Note that other errors cannot be handled via exceptions, e.g., an exercise on an inactive contract will still result in a transaction abort.

User-defined Exceptions

Users can define their own exception types which can be thrown and caught. The definition looks similar to templates, and just like with templates, the definition produces a record type of the given name as well as instances to make that type throwable and catchable.

In addition to the record fields, exceptions also need to define a `message` function.

```
exception MyException
  with
    field1 : Int
    field2 : Text
  where
    message "MyException(" <> show field1 <> ", " <> show field2 <> ")"
```

Throw Exceptions

There are two ways to throw exceptions:

1. Inside of an `Action` like `Update` or `Script` you can use `throw` from `DA.Exception`. This works for any `Action` that is an instance of `ActionThrow`.
2. Outside of `ActionThrow` you can throw exceptions using `throwPure`.

If both are an option, it is generally preferable to use `throw` since it is easier to reason about when exactly the exception will get thrown.

Catch Exceptions

Exceptions are caught in try-catch blocks similar to those found in languages like Java. The `try` block defines the scope within which errors should be handled while the `catch` clauses defines which types of errors are handled and how the program should continue. If an exception gets caught, the subtransaction between the `try` and the the point where the exception is thrown is rolled back. The actions under rollback nodes are still validated, so, e.g., you can never fetch a contract that is inactive at that point or have two contracts with the same key active at the same time. However, all ledger state changes (creates, consuming exercises) are rolled back to the state before the rollback node.

Each try-catch block can have multiple `catch` clauses with the first one that applies taking precedence.

In the example below the `create` of `T` will be rolled back and the first `catch` clause applies which will create an `Error` contract.

```
        try do
          _ <- create (T p)
          throw MyException with
            field1 = 0
            field2 = "42"
        catch
          (MyException field1 field2) ->
            create Error with
              p = p
```

(continues on next page)

(continued from previous page)

```
            msg = "MyException"
          (ArithmeticError _) ->
            create Error with
              p = p
              msg = "ArithmeticError"
```

1.43.2.13 Reference: Interfaces

In Daml, an interface defines an abstract type together with a behavior specified by its view type, method signatures, and choices. For a template to conform to this interface, there must be a corresponding `interface instance` definition where all the methods of the interface (including the special `view` method) are implemented. This allows decoupling such behavior from its implementation, so other developers can write applications in terms of the interface instead of the concrete template.

Configuration

In order to use this new feature, your Daml project needs to explicitly target Daml-LF version `1.15` or higher which is specified by adding the following line to the project's `daml.yaml` file:

```
build-options: [--target=1.15]
```

If using Canton, the protocol version of the domain should be 4 or higher, see *Canton protocol version* for more details.

Interface Declaration

An interface declaration is somewhat similar to a template declaration.

Interface Name

```
interface MyInterface where
```

This is the name of the interface.
It's preceded by the keyword `interface` and followed by the keyword `where`.
It must begin with a capital letter, like any other type name.

Implicit abstract type

Whenever an interface is defined, an abstract type is defined with the same name. Abstract here means:

- Values of this type cannot be created using a data constructor. Instead, they are constructed by applying the function `toInterface` to a template value.
- Values of this type cannot be inspected directly via case analysis. Instead, use functions such as `fromInterface`.
- See *Interface Functions* for more information on these and other functions for interacting with interface values.

An interface value carries inside it the type and parameters of the template value from which it was constructed.

As for templates, the existence of a local binding `b` of type `I`, where `I` is an interface does not necessarily imply the existence on the ledger of a contract with the template type and parameters used to construct `b`. This can only be assumed if `b` the result of a fetch from the ledger within the same transaction.

Interface Methods

```
method1 : Party
method2 : Int
method3 : Bool -> Int -> Int -> Int
```

An interface may define any number of methods.

A method definition consists of the method name and the method type, separated by a single colon `:`. The name of the method must be a valid identifier beginning with a lowercase letter or an underscore.

A method definition introduces a top level function of the same name:

- If the interface is called `I`, the method is called `m`, and the method type is `M` (which might be a function type), this introduces the function `m : I -> M`:

```
func1 : MyInterface -> Party
func1 = method1

func2 : MyInterface -> Int
func2 = method2

func3 : MyInterface -> Bool -> Int -> Int -> Int
func3 = method3
```

- The first argument's type `I` means that the function can only be applied to values of the interface type `I` itself. Methods cannot be applied to template values, even if there exists an `interface instance` of `I` for that template. To use an interface method on a template value, first convert it using the `toInterface` function.
- Applying the function to such argument results in a value of type `M`, corresponding to the implementation of `m` in the interface instance of `I` for the underlying template type `t` (the type of the template value from which the interface value was constructed).

One special method, `view`, must be defined for the viewtype. (see *Interface View Type* below).

Interface View Type

```
data MyInterfaceViewType =
  MyInterfaceViewType { name : Text, value : Int }
```

```
  viewtype MyInterfaceViewType
```

All interface instances must implement a special `view` method which returns a value of the type declared by `viewtype`.
The type must be a record.
This type is returned by subscriptions on interfaces.

Interface Choices

```
  choice MyChoice : (ContractId MyInterface, Int)
    with
      argument1 : Bool
      argument2 : Int
    controller method1 this
    do
      let n0 = method2 this
      let n1 = method3 this argument1 argument2 n0
      pure (self, n1)

  nonconsuming choice MyNonConsumingChoice : Int
    controller method1 this
    do
      pure $ method2 this
```

Interface choices work in a very similar way to template choices. Any contract of a template type for which an interface instance exists will grant the choice to the controlling party.
Interface choices can only be exercised on values of the corresponding interface type. To exercise an interface choice on a template value, first convert it using the `toInterface` function.
Interface methods can be used to define the controller of a choice (e.g. `method1`) as well as the actions that run when the choice is *exercised* (e.g. `method2` and `method3`).
As for template choices, the `choice` keyword can be optionally prefixed with the `nonconsuming` keyword to specify that the contract will not be consumed when the choice is exercised. If not specified, the choice will be `consuming`. Note that the `preconsuming` and `postconsuming` qualifiers are not supported on interface choices.
See *Reference: Choices* for full reference information, but note that controller-first syntax is not supported for interface choices.

Empty Interfaces

```
data EmptyInterfaceView = EmptyInterfaceView {}

interface YourInterface where
  viewtype EmptyInterfaceView
```

It is possible (though not necessarily useful) to define an interface without methods, precondition or choices. However, a view type must always be defined, though it can be set to unit.

Interface Instances

For context, a simple template definition:

```
template NameOfTemplate
  with
    field1 : Party
    field2 : Int
  where
    signatory field1
```

`interface instance` clause

```
    interface instance MyInterface for NameOfTemplate where
      view = MyInterfaceViewType "NameOfTemplate" 100
      method1 = field1
      method2 = field2
      method3 False _ _ = 0
      method3 True x y
        | x > 0 = x + y
        | otherwise = y
```

To make a template an instance of an existing interface, an `interface instance` clause must be defined in the template declaration.
The template of the clause must match the enclosing declaration. In other words, a template `T` declaration can only contain `interface instance` clauses where the template is `T`.
The clause must start with the keywords `interface instance`, followed by the name of the interface, then the keyword `for` and the name of the template, and finally the keyword `where`, which introduces a block where **all** the methods of the interface must be implemented.
Within the clause, there's an implicit local binding `this` referring to the contract on which the method is applied, which has the type of the template's data record. The template parameters of this contract are also in scope.
Method implementations can be defined using the same syntax as for top level functions, including pattern matches and guards (e.g. `method3`).
Each method implementation must return the same type as specified for that method in the interface declaration.
The implementation of the special `view` method must return the type specified as the `viewtype` in the interface declaration.

`interface instance` clause in the interface

```
interface MyNewInterface where
  viewtype EmptyInterfaceView
  myNewMethod1 : Party
  myNewMethod2 : Int -> Int

  choice MyNewChoice : Int
    controller myNewMethod1 this
    do pure $ myNewMethod2 this 0

  interface instance MyNewInterface for NameOfTemplate where
    view = EmptyInterfaceView
    myNewMethod1 = field1
    myNewMethod2 x = field2 `min` x
```

To make an *existing* template an instance of a new interface, the `interface instance` clause must be defined in the *interface* declaration.
In this case, the *interface* of the clause must match the enclosing declaration. In other words, an interface `I` declaration can only contain `interface instance` clauses where the interface is `I`.
All other rules for `interface instance` clauses are the same whether the enclosing declaration is a template or an interface. In particular, the implicit local binding `this` always has the type of the *template*'s record.

Empty `interface instance` clause

If the interface has no methods, the interface instance only needs to implement the `view` method:

```
  interface instance YourInterface for NameOfTemplate where
    view = EmptyInterfaceView
```

Interface Functions

`interfaceTypeRep`

Type	`HasInterfaceTypeRep i => i -> TemplateTypeRep`
Instantiated Type	`MyInterface -> TemplateTypeRep`
Notes	The value of the resulting `TemplateTypeRep` indicates what template was used to construct the interface value.

toInterface

Type	`forall i t.`
	`HasToInterface t i =>`
	`t -> i`

Instantiated Type	`MyTemplate -> MyInterface`
Notes	Converts a template value into an interface value.

fromInterface

Type	`HasFromInterface t i =>` `i -> Optional t`
Instantiated Type	`MyInterface -> Optional MyTemplate`
Notes	Attempts to convert an interface value back into a template value. The result is `None` if the expected template type doesn't match the underlying template type used to construct the contract.

toInterfaceContractId

Type	`forall i t.`
	`HasToInterface t i =>`
	`ContractId t -> ContractId i`

Instantiated Type	`ContractId MyTemplate -> ContractId MyInterface`
Notes	Converts a template Contract ID into an Interface Contract ID.

fromInterfaceContractId

Type	`forall t i. HasFromInterface t i => ContractId i -> ContractId t`
Instantiated Type	`ContractId MyInterface -> ContractId MyTemplate`
Notes	Converts an interface contract id into a template contract id. This function does not verify that the given contract id actually points to a contract of the resulting type; if that is not the case, a subsequent `fetch`, `exercise` or `archive` will fail. Therefore, this should only be used when the underlying contract is known to be of the resulting type, or when the result is immediately used by a `fetch`, `exercise` or `archive` action and a transaction failure is the desired behavior in case of mismatch. In all other cases, consider using `fetchFromInterface` instead.

coerceInterfaceContractId

Type	`forall j i. (HasInterfaceTypeRep i, HasInterfaceTypeRep j) => ContractId i -> ContractId j`
Instantiated Type	`ContractId SourceInterface -> ContractId TargetInterface`
Notes	Converts an interface contract id into a contract id of a different interface. This function does not verify that the given contract id actually points to a contract of the resulting type; if that is not the case, a subsequent `fetch`, `exercise` or `archive` will fail. Therefore, this should only be used when the underlying contract is known to be of the resulting type, or when the result is immediately used by a `fetch`, `exercise` or `archive` action and a transaction failure is the desired behavior in case of mismatch.

`fetchFromInterface`

Type	`forall t i. (HasFromInterface t i, HasFetch i) => ContractId i -> Update (Optional (ContractId t, t))`
Instantiated Type	`ContractId MyInterface -> Update (Optional (ContractId MyTemplate, MyTemplate))`
Notes	Attempts to fetch and convert an interface contract id into a template, returning both the converted contract and its contract id if the conversion is successful, or `None` otherwise.

Required Interfaces

```
interface OurInterface requires MyInterface, YourInterface where
  viewtype EmptyInterfaceView
```

An interface can depend on other interfaces. These are specified with the `requires` keyword after the interface name but before the `where` keyword, separated by commas.
For an interface declaration to be valid, its list of required interfaces must be transitively closed. In other words, an interface `I` cannot require an interface `J` without also explicitly requiring all the interfaces required by `J`. The order, however, is irrelevant.
For example, given

```
interface Shape where
  viewtype EmptyInterfaceView

interface Rectangle requires Shape where
  viewtype EmptyInterfaceView
```

This declaration for interface `Square` would cause a compiler error

```
-- Compiler error! "Interface Square is missing requirement [Shape]"
interface Square requires Rectangle where
  viewtype EmptyInterfaceView
```

Explicitly adding `Shape` to the required interfaces fixes the error

```
interface Square requires Rectangle, Shape where
  viewtype EmptyInterfaceView
```

For a template `T` to be a valid `interface instance` of an interface `I`, `T` must also be an `interface instance` of each of the interfaces required by `I`.

Interface Functions

Function	Notes
`toInterface`	Can also be used to convert an interface value to one of its required interfaces.
`fromInterface`	Can also be used to convert a value of an interface type to one of its requiring interfaces.
`toInterfaceContractId`	Can also be used to convert an interface contract id into a contract id of one of its required interfaces.
`fromInterfaceContractId`	Can also be used to convert an interface contract id into a contract id of one of its requiring interfaces.
`fetchFromInterface`	Can also be used to fetch and convert an interface contract id into a contract and contract id of one of its requiring interfaces.

1.43.3 The standard library

The Daml standard library is a collection of Daml modules that are bundled with the SDK, and can be used to implement Daml applications.

The *Prelude* module is imported automatically in every Daml module. Other modules must be imported manually, just like your own project's modules. For example:

```
import DA.Optional
import DA.Time
```

Here is a complete list of modules in the standard library:

1.43.3.1 Prelude

The pieces that make up the Daml language.

Typeclasses

class *Action* m => *CanAssert* m **where**

Constraint that determines whether an assertion can be made in this context.

assertFail : *Text* -> m t

Abort since an assertion has failed. In an Update, Scenario, Script, or Trigger context this will throw an AssertionFailed exception. In an `Either Text` context, this will return the message as an error.

instance *CanAssert Scenario*

instance *CanAssert Update*

instance *CanAssert* (*Either Text*)

class *HasInterfaceTypeRep* i **where**

(Daml-LF >= 1.15) Exposes the `interfaceTypeRep` function. Available only for interfaces.

class *HasToInterface* t i **where**

(Daml-LF >= 1.15) Exposes the `toInterface` and `toInterfaceContractId` functions.

class *HasFromInterface* t i **where**

(Daml-LF >= 1.15) Exposes `fromInterface` and `fromInterfaceContractId` functions.

fromInterface : i -> *Optional* t
(Daml-LF >= 1.15) Attempt to convert an interface value back into a template value. A `None` indicates that the expected template type doesn't match the underyling template type for the interface value.
For example, `fromInterface @MyTemplate value` will try to convert the interface value `value` into the template type `MyTemplate`.

class *HasInterfaceView* i v **where**

_view : i -> v

class *HasTime* m **where**

The `HasTime` class is for where the time is available: `Scenario` and `Update`.

getTime : *HasCallStack* => m *Time*
Get the current time.

instance *HasTime Scenario*

instance *HasTime Update*

class *Action* m => *CanAbort* m **where**

The `CanAbort` class is for `Action` s that can be aborted.

abort : *Text* -> m a
Abort the current action with a message.

instance *CanAbort Scenario*

instance *CanAbort Update*

instance *CanAbort* (*Either Text*)

class *HasSubmit* m cmds **where**

submit : *HasCallStack* => *Party* -> cmds a -> m a
`submit p cmds` submits the commands `cmds` as a single transaction from party `p` and returns the value returned by `cmds`.
If the transaction fails, `submit` also fails.

submitMustFail : *HasCallStack* => *Party* -> cmds a -> m ()
`submitMustFail p cmds` submits the commands `cmds` as a single transaction from party `p`.
It only succeeds if the submitting the transaction fails.

instance *HasSubmit Scenario Update*

class *Functor* f => *Applicative* f **where**

pure : a -> f a
Lift a value.

(<*>) : f (a -> b) -> f a -> f b
Sequentially apply the function.
A few functors support an implementation of `<*>` that is more efficient than the default one.

liftA2 : (a -> b -> c) -> f a -> f b -> f c
Lift a binary function to actions.
Some functors support an implementation of `liftA2` that is more efficient than the default one. In particular, if `fmap` is an expensive operation, it is likely better to use `liftA2` than to `fmap` over the structure and then use `<*>`.

(*>) : f a -> f b -> f b
Sequence actions, discarding the value of the first argument.

(<*) : f a -> f b -> f a
Sequence actions, discarding the value of the second argument.

instance *Applicative* ((->) r)

instance *Applicative* (*State* s)

instance *Applicative Down*

instance *Applicative Scenario*

instance *Applicative Update*

instance *Applicative Optional*

instance *Applicative Formula*

instance *Applicative NonEmpty*

instance *Applicative* (*Validation* err)

instance *Applicative* (*Either* e)

instance *Applicative ([])*

class *Applicative* m => *Action* m **where**

(>>=) : m a -> (a -> m b) -> m b
Sequentially compose two actions, passing any value produced by the first as an argument to the second.

instance *Action* ((->) r)

instance *Action* (*State* s)

instance *Action Down*

instance *Action Scenario*

instance *Action Update*

instance *Action Optional*

instance *Action Formula*

instance *Action NonEmpty*

instance *Action* (*Either* e)

instance *Action ([])*

class *Action* m => *ActionFail* m **where**

This class exists to desugar pattern matches in do-notation. Polymorphic usage, or calling `fail` directly, is not recommended. Instead consider using `CanAbort`.

fail : *Text* -> m a
Fail with an error message.

instance *ActionFail Scenario*

instance *ActionFail Update*

instance *ActionFail Optional*

instance *ActionFail* (*Either Text*)

instance *ActionFail ([])*

class *Semigroup* a **where**

The class of semigroups (types with an associative binary operation).

(<>) : a -> a -> a
An associative operation.

instance *Ord* k => *Semigroup* (*Map* k v)

instance *Semigroup* (*TextMap* b)

instance *Semigroup All*

instance *Semigroup Any*

instance *Semigroup* (*Endo* a)

instance *Multiplicative* a => *Semigroup* (*Product* a)

instance *Additive* a => *Semigroup* (*Sum* a)

instance *Semigroup* (*NonEmpty* a)

instance *Ord* a => *Semigroup* (*Max* a)

instance *Ord* a => *Semigroup* (*Min* a)

instance *Ord* k => *Semigroup* (*Set* k)

instance *Semigroup* (*Validation* err a)

instance *Semigroup Ordering*

instance *Semigroup Text*

instance *Semigroup* [a]

class *Semigroup* a => *Monoid* a **where**

The class of monoids (types with an associative binary operation that has an identity).

mempty : a
Identity of `(<>)`

mconcat : [a] -> a
Fold a list using the monoid. For example using `mconcat` on a list of strings would concatenate all strings to one lone string.

instance *Ord* k => *Monoid* (*Map* k v)

instance *Monoid* (*TextMap* b)

instance *Monoid All*

instance *Monoid Any*

instance *Monoid* (*Endo* a)

instance *Multiplicative* a => *Monoid* (*Product* a)

instance *Additive* a => *Monoid* (*Sum* a)

instance *Ord* k => *Monoid* (*Set* k)

instance *Monoid Ordering*

instance *Monoid Text*

instance *Monoid* [a]

class *HasSignatory* t **where**

Exposes `signatory` function. Part of the `Template` constraint.

signatory : t -> [*Party*]
The signatories of a contract.

class *HasObserver* t **where**

Exposes `observer` function. Part of the `Template` constraint.

observer : t -> [*Party*]
The observers of a contract.

class *HasEnsure* t **where**

Exposes `ensure` function. Part of the `Template` constraint.

ensure : t -> *Bool*
A predicate that must be true, otherwise contract creation will fail.

class *HasAgreement* t **where**

Exposes `agreement` function. Part of the `Template` constraint.

agreement : t -> *Text*
The agreement text of a contract.

class *HasCreate* t **where**

Exposes `create` function. Part of the `Template` constraint.

create : t -> *Update* (*ContractId* t)
Create a contract based on a template `t`.

class *HasFetch* t **where**

Exposes `fetch` function. Part of the `Template` constraint.

fetch : *ContractId* t -> *Update* t
Fetch the contract data associated with the given contract ID. If the `ContractId t` supplied is not the contract ID of an active contract, this fails and aborts the entire transaction.

class *HasSoftFetch* t **where**

Exposes `softFetch` function

class *HasSoftExercise* t c r **where**

class *HasArchive* t **where**

Exposes `archive` function. Part of the `Template` constraint.

archive : *ContractId* t -> *Update* ()
Archive the contract with the given contract ID.

class *HasTemplateTypeRep* t **where**

Exposes `templateTypeRep` function in Daml-LF 1.7 or later. Part of the `Template` constraint.

class *HasToAnyTemplate* t **where**

Exposes `toAnyTemplate` function in Daml-LF 1.7 or later. Part of the `Template` constraint.

class *HasFromAnyTemplate* t **where**

Exposes `fromAnyTemplate` function in Daml-LF 1.7 or later. Part of the `Template` constraint.

class *HasExercise* t c r **where**

Exposes `exercise` function. Part of the `Choice` constraint.

exercise : *ContractId* t -> c -> *Update* r
Exercise a choice on the contract with the given contract ID.

class *HasDynamicExercise* t c r **where**

class *HasChoiceController* t c **where**

Exposes `choiceController` function. Part of the `Choice` constraint.

class *HasChoiceObserver* t c **where**

Exposes `choiceObserver` function. Part of the `Choice` constraint.

class *HasExerciseGuarded* t c r **where**

(1.dev only) Exposes `exerciseGuarded` function. Only available for interface choices.

exerciseGuarded : (t -> *Bool*) -> *ContractId* t -> c -> *Update* r
(1.dev only) Exercise a choice on the contract with the given contract ID, only if the predicate returns `True`.

class *HasToAnyChoice* t c r **where**

Exposes `toAnyChoice` function for Daml-LF 1.7 or later. Part of the `Choice` constraint.

class *HasFromAnyChoice* t c r **where**

Exposes `fromAnyChoice` function for Daml-LF 1.7 or later. Part of the `Choice` constraint.

class *HasKey* t k **where**

Exposes `key` function. Part of the `TemplateKey` constraint.

key : t -> k
The key of a contract.

class *HasLookupByKey* t k **where**

Exposes `lookupByKey` function. Part of the `TemplateKey` constraint.

lookupByKey : k -> *Update* (*Optional* (*ContractId* t))
Look up the contract ID `t` associated with a given contract key `k`.
You must pass the `t` using an explicit type application. For instance, if you want to look up a contract of template `Account` by its key `k`, you must call `lookupByKey @Account k`.

class *HasFetchByKey* t k **where**

Exposes `fetchByKey` function. Part of the `TemplateKey` constraint.

fetchByKey : k -> *Update* (*ContractId* t, t)
Fetch the contract ID and contract data associated with a given contract key.
You must pass the `t` using an explicit type application. For instance, if you want to fetch a contract of template `Account` by its key `k`, you must call `fetchByKey @Account k`.

class *HasMaintainer* t k **where**

Exposes `maintainer` function. Part of the `TemplateKey` constraint.

class *HasToAnyContractKey* t k **where**

Exposes `toAnyContractKey` function in Daml-LF 1.7 or later. Part of the `TemplateKey` constraint.

class *HasFromAnyContractKey* t k **where**

Exposes `fromAnyContractKey` function in Daml-LF 1.7 or later. Part of the `TemplateKey` constraint.

class *HasExerciseByKey* t k c r **where**

Exposes `exerciseByKey` function.

class *IsParties* a **where**

Accepted ways to specify a list of parties: either a single party, or a list of parties.

toParties : a -> [*Party*]
Convert to list of parties.

instance *IsParties Party*

instance *IsParties* (*Optional Party*)

instance *IsParties* (*NonEmpty Party*)

instance *IsParties* (*Set Party*)

instance *IsParties* [*Party*]

class *Functor* f **where**

A `Functor` is a typeclass for things that can be mapped over (using its `fmap` function. Examples include `Optional`, `[]` and `Update`).

fmap : (a -> b) -> f a -> f b

`fmap` takes a function of type `a -> b`, and turns it into a function of type `f a -> f b`, where `f` is the type which is an instance of `Functor`.
For example, `map` is an `fmap` that only works on lists. It takes a function `a -> b` and a `[a]`, and returns a `[b]`.

(<$) : a -> f b -> f a

Replace all locations in the input `f b` with the same value `a`. The default definition is `fmap . const`, but you can override this with a more efficient version.

class *Eq* a **where**

The `Eq` class defines equality (==) and inequality (/=). All the basic datatypes exported by the "Prelude" are instances of `Eq`, and `Eq` may be derived for any datatype whose constituents are also instances of `Eq`.

Usually, == is expected to implement an equivalence relationship where two values comparing equal are indistinguishable by "public" functions, with a "public" function being one not allowing to see implementation details. For example, for a type representing non-normalised natural numbers modulo 100, a "public" function doesn't make the difference between 1 and 201. It is expected to have the following properties:

Reflexivity: `x == x = True`

Symmetry: `x == y = y == x`

Transitivity: if `x == y && y == z = True`, then `x == z = True`

Substitutivity: if `x == y = True` and `f` is a "public" function whose return type is an instance of `Eq`, then `f x == f y = True`

Negation: `x /= y = not (x == y)`

Minimal complete definition: either == or /=.

(==) : a -> a -> *Bool*

(/=) : a -> a -> *Bool*

instance (*Eq* a, *Eq* b) => *Eq* (*Either* a b)

instance *Eq BigNumeric*

instance *Eq Bool*

instance *Eq Int*

instance *Eq* (*Numeric* n)

instance *Eq Ordering*

instance *Eq RoundingMode*

instance *Eq Text*

instance *Eq* a => *Eq* [a]

instance *Eq* ()

instance (*Eq* a, *Eq* b) => *Eq* (a, b)

instance (*Eq* a, *Eq* b, *Eq* c) => *Eq* (a, b, c)

instance (*Eq* a, *Eq* b, *Eq* c, *Eq* d) => *Eq* (a, b, c, d)

instance (*Eq* a, *Eq* b, *Eq* c, *Eq* d, *Eq* e) => *Eq* (a, b, c, d, e)

instance (*Eq* a, *Eq* b, *Eq* c, *Eq* d, *Eq* e, *Eq* f) => *Eq* (a, b, c, d, e, f)

instance (*Eq* a, *Eq* b, *Eq* c, *Eq* d, *Eq* e, *Eq* f, *Eq* g) => *Eq* (a, b, c, d, e, f, g)

instance (*Eq* a, *Eq* b, *Eq* c, *Eq* d, *Eq* e, *Eq* f, *Eq* g, *Eq* h) => *Eq* (a, b, c, d, e, f, g, h)

instance (*Eq* a, *Eq* b, *Eq* c, *Eq* d, *Eq* e, *Eq* f, *Eq* g, *Eq* h, *Eq* i) => *Eq* (a, b, c, d, e, f, g, h, i)

instance (*Eq* a, *Eq* b, *Eq* c, *Eq* d, *Eq* e, *Eq* f, *Eq* g, *Eq* h, *Eq* i, *Eq* j) => *Eq* (a, b, c, d, e, f, g, h, i, j)

instance (*Eq* a, *Eq* b, *Eq* c, *Eq* d, *Eq* e, *Eq* f, *Eq* g, *Eq* h, *Eq* i, *Eq* j, *Eq* k) => *Eq* (a, b, c, d, e, f, g, h, i, j, k)

instance (*Eq* a, *Eq* b, *Eq* c, *Eq* d, *Eq* e, *Eq* f, *Eq* g, *Eq* h, *Eq* i, *Eq* j, *Eq* k, *Eq* l) => *Eq* (a, b, c, d, e, f, g, h, i, j, k, l)

instance (*Eq* a, *Eq* b, *Eq* c, *Eq* d, *Eq* e, *Eq* f, *Eq* g, *Eq* h, *Eq* i, *Eq* j, *Eq* k, *Eq* l, *Eq* m) => *Eq* (a, b, c, d, e, f, g, h, i, j, k, l, m)

instance (*Eq* a, *Eq* b, *Eq* c, *Eq* d, *Eq* e, *Eq* f, *Eq* g, *Eq* h, *Eq* i, *Eq* j, *Eq* k, *Eq* l, *Eq* m, *Eq* n) => *Eq* (a, b, c, d, e, f, g, h, i, j, k, l, m, n)

instance (*Eq* a, *Eq* b, *Eq* c, *Eq* d, *Eq* e, *Eq* f, *Eq* g, *Eq* h, *Eq* i, *Eq* j, *Eq* k, *Eq* l, *Eq* m, *Eq* n, *Eq* o) => *Eq* (a, b, c, d, e, f, g, h, i, j, k, l, m, n, o)

class *Eq* a => *Ord* a **where**

The `Ord` class is used for totally ordered datatypes.

Instances of `Ord` can be derived for any user-defined datatype whose constituent types are in `Ord`. The declared order of the constructors in the data declaration determines the ordering in derived `Ord` instances. The `Ordering` datatype allows a single comparison to determine the precise ordering of two objects.

The Haskell Report defines no laws for `Ord`. However, `<=` is customarily expected to implement a non-strict partial order and have the following properties:

Transitivity: if `x <= y && y <= z = True`, then `x <= z = True`

Reflexivity: `x <= x = True`

Antisymmetry: if `x <= y && y <= x = True`, then `x == y = True`

Note that the following operator interactions are expected to hold:

1. `x >= y = y <= x`
2. `x < y = x <= y && x /= y`
3. `x > y = y < x`
4. `x < y = compare x y == LT`
5. `x > y = compare x y == GT`
6. `x == y = compare x y == EQ`
7. `min x y == if x <= y then x else y` = 'True'
8. `max x y == if x >= y then x else y` = 'True'

Minimal complete definition: either `compare` or `<=`. Using `compare` can be more efficient for complex types.

compare : a -> a -> *Ordering*

(<) : a -> a -> *Bool*

(<=) : a -> a -> *Bool*

(>) : a -> a -> *Bool*

(>=) : a -> a -> *Bool*

max : a -> a -> a

min : a -> a -> a

instance (*Ord* a, *Ord* b) => *Ord* (*Either* a b)

instance *Ord BigNumeric*

instance *Ord Bool*

instance *Ord Int*

instance *Ord* (*Numeric* n)

instance *Ord Ordering*

instance *Ord RoundingMode*

instance *Ord Text*

instance *Ord* a => *Ord* [a]

instance *Ord* ()

instance (*Ord* a, *Ord* b) => *Ord* (a, b)

instance (*Ord* a, *Ord* b, *Ord* c) => *Ord* (a, b, c)

instance (*Ord* a, *Ord* b, *Ord* c, *Ord* d) => *Ord* (a, b, c, d)

instance (*Ord* a, *Ord* b, *Ord* c, *Ord* d, *Ord* e) => *Ord* (a, b, c, d, e)

instance (*Ord* a, *Ord* b, *Ord* c, *Ord* d, *Ord* e, *Ord* f) => *Ord* (a, b, c, d, e, f)

instance (*Ord* a, *Ord* b, *Ord* c, *Ord* d, *Ord* e, *Ord* f, *Ord* g) => *Ord* (a, b, c, d, e, f, g)

instance (*Ord* a, *Ord* b, *Ord* c, *Ord* d, *Ord* e, *Ord* f, *Ord* g, *Ord* h) => *Ord* (a, b, c, d, e, f, g, h)

instance (*Ord* a, *Ord* b, *Ord* c, *Ord* d, *Ord* e, *Ord* f, *Ord* g, *Ord* h, *Ord* i) => *Ord* (a, b, c, d, e, f, g, h, i)

instance (*Ord* a, *Ord* b, *Ord* c, *Ord* d, *Ord* e, *Ord* f, *Ord* g, *Ord* h, *Ord* i, *Ord* j) => *Ord* (a, b, c, d, e, f, g, h, i, j)

instance (*Ord* a, *Ord* b, *Ord* c, *Ord* d, *Ord* e, *Ord* f, *Ord* g, *Ord* h, *Ord* i, *Ord* j, *Ord* k) => *Ord* (a, b, c, d, e, f, g, h, i, j, k)

instance (*Ord* a, *Ord* b, *Ord* c, *Ord* d, *Ord* e, *Ord* f, *Ord* g, *Ord* h, *Ord* i, *Ord* j, *Ord* k, *Ord* l) => *Ord* (a, b, c, d, e, f, g, h, i, j, k, l)

instance (*Ord* a, *Ord* b, *Ord* c, *Ord* d, *Ord* e, *Ord* f, *Ord* g, *Ord* h, *Ord* i, *Ord* j, *Ord* k, *Ord* l, *Ord* m) => *Ord* (a, b, c, d, e, f, g, h, i, j, k, l, m)

instance (*Ord* a, *Ord* b, *Ord* c, *Ord* d, *Ord* e, *Ord* f, *Ord* g, *Ord* h, *Ord* i, *Ord* j, *Ord* k, *Ord* l, *Ord* m, *Ord* n) => *Ord* (a, b, c, d, e, f, g, h, i, j, k, l, m, n)

instance (*Ord* a, *Ord* b, *Ord* c, *Ord* d, *Ord* e, *Ord* f, *Ord* g, *Ord* h, *Ord* i, *Ord* j, *Ord* k, *Ord* l, *Ord* m, *Ord* n, *Ord* o) => *Ord* (a, b, c, d, e, f, g, h, i, j, k, l, m, n, o)

class *NumericScale* n **where**

Is this a valid scale for the `Numeric` type?

This typeclass is used to prevent the creation of Numeric values with too large a scale. The scale controls the number of digits available after the decimal point, and it must be between 0 and 37 inclusive.

Thus the only available instances of this typeclass are `NumericScale 0` through `NumericScale 37`. This cannot be extended without additional compiler and runtime support. You cannot implement a custom instance of this typeclass.

If you have an error message in your code of the form "No instance for `(NumericScale n)`", this is probably caused by having a numeric literal whose scale cannot be inferred by the compiler. You can usually fix this by adding a type signature to the definition, or annotating the numeric literal directly (for example, instead of writing `3.14159` you can write `(3.14159 : Numeric 5)`).

numericScale : proxy n -> *Int*

Get the scale of a `Numeric` as an integer. For example, `numericScale (3.14159 : Numeric 5)` equals `5`.

instance *NumericScale* 0

instance *NumericScale* 1

instance *NumericScale* 10

instance *NumericScale* 11

instance *NumericScale* 12

instance *NumericScale* 13

instance *NumericScale* 14

instance *NumericScale* 15

instance *NumericScale* 16

instance *NumericScale* 17

instance *NumericScale* 18

instance *NumericScale* 19

instance *NumericScale* 2

instance *NumericScale* 20

instance *NumericScale* 21

instance *NumericScale* 22

instance *NumericScale* 23

instance *NumericScale* 24

instance *NumericScale* 25

instance *NumericScale* 26

instance *NumericScale* 27

instance *NumericScale* 28

instance *NumericScale* 29

instance *NumericScale* 3

instance *NumericScale* 30

instance *NumericScale* 31

instance *NumericScale* 32

instance *NumericScale* 33

instance *NumericScale* 34

instance *NumericScale* 35

instance *NumericScale* 36

instance *NumericScale* 37

instance *NumericScale* 4

instance *NumericScale* 5

instance *NumericScale* 6

instance *NumericScale* 7

instance *NumericScale* 8

instance *NumericScale* 9

class *IsNumeric* t **where**

Types that can be represented by decimal literals in Daml.

fromNumeric : *NumericScale* m => *Numeric* m -> t
Convert from `Numeric`. Raises an error if the number can't be represented exactly in the target type.

fromBigNumeric : *BigNumeric* -> t
Convert from `BigNumeric`. Raises an error if the number can't be represented exactly in the target type.

instance *IsNumeric BigNumeric*

instance *NumericScale* n => *IsNumeric* (*Numeric* n)

class *Bounded* a **where**

Use the `Bounded` class to name the upper and lower limits of a type.

You can derive an instance of the `Bounded` class for any enumeration type. `minBound` is the first constructor listed in the `data` declaration and `maxBound` is the last.

You can also derive an instance of `Bounded` for single-constructor data types whose constituent types are in `Bounded`.

`Ord` is not a superclass of `Bounded` because types that are not totally ordered can still have upper and lower bounds.

minBound : a

maxBound : a

instance *Bounded Bool*

instance *Bounded Int*

class *Enum* a **where**

Use the `Enum` class to define operations on sequentially ordered types: that is, types that can be enumerated. `Enum` members have defined successors and predecessors, which you can get with the `succ` and `pred` functions.

Types that are an instance of class `Bounded` as well as `Enum` should respect the following laws:

Both `succ maxBound` and `pred minBound` should result in a runtime error.
`fromEnum` and `toEnum` should give a runtime error if the result value is not representable in the result type. For example, `toEnum 7 : Bool` is an error.
`enumFrom` and `enumFromThen` should be defined with an implicit bound, like this:

```
enumFrom     x   = enumFromTo     x maxBound
enumFromThen x y = enumFromThenTo x y bound
    where
        bound | fromEnum y >= fromEnum x = maxBound
              | otherwise                = minBound
```

succ : a -> a
Returns the successor of the given value. For example, for numeric types, `succ` adds 1.
If the type is also an instance of `Bounded`, `succ maxBound` results in a runtime error.

pred : a -> a
Returns the predecessor of the given value. For example, for numeric types, `pred` subtracts 1.
If the type is also an instance of `Bounded`, `pred minBound` results in a runtime error.

toEnum : *Int* -> a
Convert a value from an `Int` to an `Enum` value: ie, `toEnum i` returns the item at the `i` th position of (the instance of) `Enum`

fromEnum : a -> *Int*
Convert a value from an `Enum` value to an `Int`: ie, returns the `Int` position of the element within the `Enum`.
If `fromEnum` is applied to a value that's too large to fit in an `Int`, what is returned is up to your implementation.

enumFrom : a -> [a]
Return a list of the `Enum` values starting at the `Int` position. For example:

```
enumFrom 6 : [Int] = [6,7,8,9,...,maxBound : Int]
```

enumFromThen : a -> a -> [a]
Returns a list of the `Enum` values with the first value at the first `Int` position, the second value at the second `Int` position, and further values with the same distance between them.
For example:

```
enumFromThen 4 6 : [Int] = [4,6,8,10...]
enumFromThen 6 2 : [Int] = [6,2,-2,-6,...,minBound :: Int]
```

enumFromTo : a -> a -> [a]
Returns a list of the `Enum` values with the first value at the first `Int` position, and the last value at the last `Int` position.
This is what's behind the language feature that lets you write `[n,m..]`.
For example:

```
enumFromTo 6 10 : [Int] = [6,7,8,9,10]
```

enumFromThenTo : a -> a -> a -> [a]
Returns a list of the `Enum` values with the first value at the first `Int` position, the second value at the second `Int` position, and further values with the same distance between them, with the final value at the final `Int` position.
This is what's behind the language feature that lets you write `[n,n'..m]`.
For example:

```
enumFromThenTo 4 2 -6 : [Int] = [4,2,0,-2,-4,-6]
enumFromThenTo 6 8 2 : [Int] = []
```

instance *Enum Bool*

instance *Enum Int*

class *Additive* a **where**

Use the `Additive` class for types that can be added. Instances have to respect the following laws:

`(+)` must be associative, ie: `(x + y) + z = x + (y + z)`
`(+)` must be commutative, ie: `x + y = y + x`
`x + aunit = x`
`negate` gives the additive inverse, ie: `x + negate x = aunit`

(+) : a -> a -> a
Add the two arguments together.

aunit : a
The additive identity for the type. For example, for numbers, this is 0.

(-) : a -> a -> a
Subtract the second argument from the first argument, ie. `x - y = x + negate y`

negate : a -> a
Negate the argument: `x + negate x = aunit`

instance *Additive BigNumeric*

instance *Additive Int*

instance *Additive* (*Numeric* n)

class *Multiplicative* a **where**

Use the `Multiplicative` class for types that can be multiplied. Instances have to respect the following laws:

`(*)` is associative, ie: `(x * y) * z = x * (y * z)`
`(*)` is commutative, ie: `x * y = y * x`
`x * munit = x`

(*) : a -> a -> a
Multipy the arguments together

munit : a
The multiplicative identity for the type. For example, for numbers, this is 1.

(^) : a -> *Int* -> a
`x ^ n` raises `x` to the power of `n`.

instance *Multiplicative BigNumeric*

instance *Multiplicative Int*

instance *Multiplicative* (*Numeric* n)

class (*Additive* a, *Multiplicative* a) => *Number* a **where**

`Number` is a class for numerical types. As well as the rules for `Additive` and `Multiplicative`, instances also have to respect the following law:

`(*)` is distributive with respect to `(+)`. That is: `a * (b + c) = (a * b) + (a * c)` and `(b + c) * a = (b * a) + (c * a)`

instance *Number BigNumeric*

instance *Number Int*

instance *Number* (*Numeric* n)

class *Signed* a **where**

The `Signed` is for the sign of a number.

signum : a -> a
Sign of a number. For real numbers, the 'signum' is either `-1` (negative), `0` (zero) or `1` (positive).

abs : a -> a
The absolute value: that is, the value without the sign.

instance *Signed BigNumeric*

instance *Signed Int*

instance *Signed* (*Numeric* n)

class *Multiplicative* a => *Divisible* a **where**

Use the `Divisible` class for types that can be divided. Instances should respect that division is the inverse of multiplication, i.e. `x * y / y` is equal to `x` whenever it is defined.

(/) : a -> a -> a
`x / y` divides `x` by `y`

instance *Divisible Int*

instance *Divisible* (*Numeric* n)

class *Divisible* a => *Fractional* a **where**

Use the `Fractional` class for types that can be divided and where the reciprocal is well defined. Instances have to respect the following laws:

When `recip x` is defined, it must be the inverse of `x` with respect to multiplication: `x * recip x = munit`
When `recip y` is defined, then `x / y = x * recip y`

recip : a -> a
Calculates the reciprocal: `recip x` is `1/x`.

instance *Fractional* (*Numeric* n)

class *Show* a **where**

Use the `Show` class for values that can be converted to a readable `Text` value.

Derived instances of `Show` have the following properties:

> The result of `show` is a syntactically correct expression that only contains constants (given the fixity declarations in force at the point where the type is declared). It only contains the constructor names defined in the data type, parentheses, and spaces. When labelled constructor fields are used, braces, commas, field names, and equal signs are also used.
> If the constructor is defined to be an infix operator, then `showsPrec` produces infix applications of the constructor.
> If the precedence of the top-level constructor in `x` is less than `d` (associativity is ignored), the representation will be enclosed in parentheses. For example, if `d` is `0` then the result is never surrounded in parentheses; if `d` is `11` it is always surrounded in parentheses, unless it is an atomic expression.
> If the constructor is defined using record syntax, then `show` will produce the record-syntax form, with the fields given in the same order as the original declaration.

showsPrec : *Int* -> a -> *ShowS*
Convert a value to a readable `Text` value. Unlike `show`, `showsPrec` should satisfy the rule `showsPrec d x r ++ s == showsPrec d x (r ++ s)`

show : a -> *Text*
Convert a value to a readable `Text` value.

showList : [a] -> *ShowS*
Allows you to show lists of values.

instance (*Show* a, *Show* b) => *Show* (*Either* a b)

instance *Show BigNumeric*

instance *Show Bool*

instance *Show Int*

instance *Show* (*Numeric* n)

instance *Show Ordering*

instance *Show RoundingMode*

instance *Show Text*

instance *Show* a => *Show* [a]

instance *Show* ()

instance (*Show* a, *Show* b) => *Show* (a, b)

instance (*Show* a, *Show* b, *Show* c) => *Show* (a, b, c)

instance (*Show* a, *Show* b, *Show* c, *Show* d) => *Show* (a, b, c, d)

instance (*Show* a, *Show* b, *Show* c, *Show* d, *Show* e) => *Show* (a, b, c, d, e)

Data Types

data *AnyChoice*

Existential choice type that can wrap an arbitrary choice.

AnyChoice

Field	Type	Description
getAnyChoice	Any	
getAnyChoiceTemplateTypeRep	*TemplateTypeRep*	

instance *Eq AnyChoice*

instance *Ord AnyChoice*

data *AnyContractKey*

Existential contract key type that can wrap an arbitrary contract key.

AnyContractKey

Field	Type	Description
getAnyContractKey	Any	
getAnyContractKeyTemplateTypeRep	*TemplateTypeRep*	

instance *Eq AnyContractKey*

instance *Ord AnyContractKey*

data *AnyTemplate*

Existential template type that can wrap an arbitrary template.

AnyTemplate

Field	Type	Description
getAnyTemplate	Any	

instance *Eq AnyTemplate*

instance *Ord AnyTemplate*

data *TemplateTypeRep*

Unique textual representation of a template Id.

TemplateTypeRep

Field	Type	Description
getTemplateType-Rep	TypeRep	

instance *Eq TemplateTypeRep*

instance *Ord TemplateTypeRep*

data *Down* a

The `Down` type can be used for reversing sorting order. For example, `sortOn (\x -> Down x.field)` would sort by descending `field`.

Down a

instance *Action Down*

instance *Applicative Down*

instance *Functor Down*

instance *Eq* a => *Eq* (*Down* a)

instance *Ord* a => *Ord* (*Down* a)

instance *Show* a => *Show* (*Down* a)

type ***Implements*** **t i** = (*HasInterfaceTypeRep* i, *HasToInterface* t i, *HasFromInterface* t i)
(Daml-LF >= 1.15) Constraint that indicates that a template implements an interface.

data *AnyException*

A wrapper for all exception types.

instance *HasFromAnyException AnyException*

instance *HasMessage AnyException*

instance *HasToAnyException AnyException*

data *ContractId* a

The `ContractId a` type represents an ID for a contract created from a template `a`. You can use the ID to fetch the contract, among other things.

instance *Eq* (*ContractId* a)

instance *Ord* (*ContractId* a)

instance *Show* (*ContractId* a)

data *Date*

The `Date` type represents a date, for example `date 2007 Apr 5`.

instance *Eq Date*

instance *Ord Date*

instance *Bounded Date*

instance *Enum Date*

instance *Show Date*

data *Map* a b

The `Map a b` type represents an associative array from keys of type `a` to values of type `b`. It uses the built-in equality for keys. Import `DA.Map` to use it.

instance *Ord* k => *Foldable* (*Map* k)

instance *Ord* k => *Monoid* (*Map* k v)

instance *Ord* k => *Semigroup* (*Map* k v)

instance *Ord* k => *Traversable* (*Map* k)

instance *Ord* k => *Functor* (*Map* k)

instance (*Ord* k, *Eq* v) => *Eq* (*Map* k v)

instance (*Ord* k, *Ord* v) => *Ord* (*Map* k v)

instance (*Show* k, *Show* v) => *Show* (*Map* k v)

data *Party*

The `Party` type represents a party to a contract.

instance *IsParties Party*

instance *IsParties* (*Optional Party*)

instance *IsParties* (*NonEmpty Party*)

instance *IsParties* (*Set Party*)

instance *IsParties* [*Party*]

instance *Eq Party*

instance *Ord Party*

instance *Show Party*

data *Scenario* a

The `Scenario` type is for simulating ledger interactions. The type `Scenario a` describes a set of actions taken by various parties during the simulated scenario, before returning a value of type `a`.

instance *CanAssert Scenario*

instance *ActionThrow Scenario*

instance *CanAbort Scenario*

instance *HasSubmit Scenario Update*

instance *HasTime Scenario*

instance *Action Scenario*

instance *ActionFail Scenario*

instance *Applicative Scenario*

instance *Functor Scenario*

data *TextMap* a

The `TextMap a` type represents an associative array from keys of type `Text` to values of type `a`.

instance *Foldable TextMap*

instance *Monoid* (*TextMap* b)

instance *Semigroup* (*TextMap* b)

instance *Traversable TextMap*

instance *Functor TextMap*

instance *Eq* a => *Eq* (*TextMap* a)

instance *Ord* a => *Ord* (*TextMap* a)

instance *Show* a => *Show* (*TextMap* a)

data *Time*

The `Time` type represents a specific datetime in UTC, for example `time (date 2007 Apr 5) 14 30 05`.

instance *Eq Time*

instance *Ord Time*

instance *Show Time*

data *Update* a

The `Update a` type represents an `Action` to update or query the ledger, before returning a value of type `a`. Examples include `create` and `fetch`.

instance *CanAssert Update*

instance *ActionCatch Update*

instance *ActionThrow Update*

instance *CanAbort Update*

instance *HasSubmit Scenario Update*

instance *HasTime Update*

instance *Action Update*

instance *ActionFail Update*

instance *Applicative Update*

instance *Functor Update*

data *Optional* a

The `Optional` type encapsulates an optional value. A value of type `Optional a` either contains a value of type `a` (represented as `Some a`), or it is empty (represented as `None`). Using `Optional` is a good way to deal with errors or exceptional cases without resorting to drastic measures such as `error`.

The `Optional` type is also an `Action`. It is a simple kind of error `Action`, where all errors are represented by `None`. A richer error `Action` could be built using the `Data.Either.Either` type.

None

Some a

instance *Foldable Optional*

instance *Action Optional*

instance *ActionFail Optional*

instance *Applicative Optional*

instance *IsParties* (*Optional Party*)

instance *Traversable Optional*

instance *Functor Optional*

instance *Eq* a => *Eq* (*Optional* a)

instance *Ord* a => *Ord* (*Optional* a)

instance *Show* a => *Show* (*Optional* a)

data *Archive*

The data type corresponding to the implicit `Archive` choice in every template.

Archive

(no fields)

instance *Eq Archive*

instance *Show Archive*

type ***Choice*** **t c r** = (*Template* t, *HasExercise* t c r, *HasToAnyChoice* t c r, *HasFromAnyChoice* t c r)
Constraint satisfied by choices.

type ***Template*** **t** = (*HasTemplateTypeRep* t, *HasToAnyTemplate* t, *HasFromAnyTemplate* t)

type ***TemplateKey*** **t k** = (*Template* t, *HasKey* t k, *HasLookupByKey* t k, *HasFetchByKey* t k, *HasMaintainer* t k, *HasToAnyContractKey* t k, *HasFromAnyContractKey* t k)
Constraint satisfied by template keys.

data *Either* a b

The `Either` type represents values with two possibilities: a value of type `Either a b` is either `Left a` or `Right b`.

The `Either` type is sometimes used to represent a value which is either correct or an error; by convention, the `Left` constructor is used to hold an error value and the `Right` constructor is used to hold a correct value (mnemonic: "right" also means "correct").

Left a

Right b

instance (*Eq* a, *Eq* b) => *Eq* (*Either* a b)

instance (*Ord* a, *Ord* b) => *Ord* (*Either* a b)

instance (*Show* a, *Show* b) => *Show* (*Either* a b)

type ***ShowS*** = *Text* -> *Text*

`showS` should represent some text, and applying it to some argument should prepend the argument to the represented text.

data *BigNumeric*

A big numeric type, capable of holding large decimal values with many digits.

`BigNumeric` represents any positive or negative decimal number with up to 2^15 digits before the decimal point, and up to 2^15 digits after the decimal point.

`BigNumeric` is not serializable, it is only intended for intermediate computation. You must round and convert `BigNumeric` to a fixed-width numeric (`Numeric n`) in order to store it in a template. The rounding operations are `round` and `div` from the `DA.BigNumeric` module. The casting operations are `fromNumeric` and `fromBigNumeric` from the `IsNumeric` typeclass.

instance *Eq BigNumeric*

instance *IsNumeric BigNumeric*

instance *Ord BigNumeric*

instance *Additive BigNumeric*

instance *Multiplicative BigNumeric*

instance *Number BigNumeric*

instance *Signed BigNumeric*

instance *Show BigNumeric*

data *Bool*

A type for Boolean values, ie `True` and `False`.

False

True

instance *Eq Bool*

instance *Ord Bool*

instance *Bounded Bool*

instance *Enum Bool*

instance *Show Bool*

type ***Decimal*** = *Numeric* 10

data *Int*

A type representing a 64-bit integer.

instance *Eq Int*

instance *Ord Int*

instance *Bounded Int*

instance *Enum Int*

instance *Additive Int*

instance *Divisible Int*

instance *Multiplicative Int*

instance *Number Int*

instance *Signed Int*

instance *Show Int*

data *Nat*

(Kind) This is the kind of type-level naturals.

data *Numeric* n

A type for fixed-point decimal numbers, with the scale being passed as part of the type.

`Numeric n` represents a fixed-point decimal number with a fixed precision of 38 (i.e. 38 digits not including a leading zero) and a scale of `n`, i.e., `n` digits after the decimal point.

`n` must be between 0 and 37 (bounds inclusive).

Examples:

```
0.01 : Numeric 2
0.0001 : Numeric 4
```

instance *Eq* (*Numeric* n)

instance *NumericScale* n => *IsNumeric* (*Numeric* n)

instance *Ord* (*Numeric* n)

instance *Additive* (*Numeric* n)

instance *Divisible* (*Numeric* n)

instance *Fractional* (*Numeric* n)

instance *Multiplicative* (*Numeric* n)

instance *Number* (*Numeric* n)

instance *Signed* (*Numeric* n)

instance *Show* (*Numeric* n)

data *Ordering*

A type for giving information about ordering: being less than (`LT`), equal to (`EQ`), or greater than (`GT`) something.

LT

EQ

GT

instance *Eq Ordering*

instance *Ord Ordering*

instance *Show Ordering*

data *RoundingMode*

Rounding modes for `BigNumeric` operations like `div` and `round` from `DA.BigNumeric`.

RoundingUp

Round away from zero.

RoundingDown

Round towards zero.

RoundingCeiling

Round towards positive infinity.

RoundingFloor

Round towards negative infinity.

RoundingHalfUp

Round towards the nearest neighbor unless both neighbors are equidistant, in which case round away from zero.

RoundingHalfDown

Round towards the nearest neighbor unless both neighbors are equidistant, in which case round towards zero.

RoundingHalfEven

Round towards the nearest neighbor unless both neighbors are equidistant, in which case round towards the even neighbor.

RoundingUnnecessary

Do not round. Raises an error if the result cannot be represented without rounding at the targeted scale.

instance *Eq RoundingMode*

instance *Ord RoundingMode*

instance *Show RoundingMode*

data *Text*

A type for text strings, that can represent any unicode code point. For example `"Hello, world"`.

instance *Eq Text*

instance *Ord Text*

instance *Show Text*

data *[]* a

A type for lists, for example `[1,2,3]`.

([])

(:) _ _

Functions

assert : *CanAssert* m => *Bool* -> m ()
Check whether a condition is true. If it's not, abort the transaction.

assertMsg : *CanAssert* m => *Text* -> *Bool* -> m ()
Check whether a condition is true. If it's not, abort the transaction with a message.

assertAfter : (*CanAssert* m, *HasTime* m) => *Time* -> m ()
Check whether the given time is in the future. If it's not, abort the transaction.

assertBefore : (*CanAssert* m, *HasTime* m) => *Time* -> m ()
Check whether the given time is in the past. If it's not, abort the transaction.

daysSinceEpochToDate : *Int* -> *Date*
Convert from number of days since epoch (i.e. the number of days since January 1, 1970) to a date.

dateToDaysSinceEpoch : *Date* -> *Int*
Convert from a date to number of days from epoch (i.e. the number of days since January 1, 1970).

interfaceTypeRep : *HasInterfaceTypeRep* i => i -> *TemplateTypeRep*
(Daml-LF >= 1.15) Obtain the `TemplateTypeRep` for the template given in the interface value.

toInterface : *HasToInterface* t i => t -> i
(Daml-LF >= 1.15) Convert a template value into an interface value. For example `toInterface @MyInterface value` converts a template `value` into a `MyInterface` type.

toInterfaceContractId : *HasToInterface* t i => *ContractId* t -> *ContractId* i
(Daml-LF >= 1.15) Convert a template contract id into an interface contract id. For example, `toInterfaceContractId @MyInterface cid`.

fromInterfaceContractId : *HasFromInterface* t i => *ContractId* i -> *ContractId* t
(Daml-LF >= 1.15) Convert an interface contract id into a template contract id. For example, `fromInterfaceContractId @MyTemplate cid`.
Can also be used to convert an interface contract id into a contract id of one of its requiring interfaces.
This function does not verify that the interface contract id actually points to a template of the given type. This means that a subsequent `fetch`, `exercise`, or `archive` may fail, if, for example, the contract id points to a contract that implements the interface but is of a different template type than expected.
Therefore, you should only use `fromInterfaceContractId` in situations where you already know that the contract id points to a contract of the right template type. You can also use it in situations where you will fetch, exercise, or archive the contract right away, when a transaction failure is the appropriate response to the contract having the wrong template type.
In all other cases, consider using `fetchFromInterface` instead.

coerceInterfaceContractId : (*HasInterfaceTypeRep* i, *HasInterfaceTypeRep* j) => *ContractId* i -> *ContractId* j
(Daml-LF >= 1.15) Convert an interface contract id into a contract id of a different interface. For example, given two interfaces `Source` and `Target`, and `cid : ContractId Source`, `coerceInterfaceContractId @Target @Source cid : ContractId Target`.
This function does not verify that the contract id actually points to a contract that implements either interface. This means that a subsequent `fetch`, `exercise`, or `archive` may fail, if, for example, the contract id points to a contract of template `A` but it was coerced into a `ContractId B` where `B` is an interface and there's no interface instance B for A.

Therefore, you should only use `coerceInterfaceContractId` in situations where you already know that the contract id points to a contract of the right type. You can also use it in situations where you will fetch, exercise, or archive the contract right away, when a transaction failure is the appropriate response to the contract having the wrong type.

fetchFromInterface : (*HasFromInterface* t i, *HasFetch* i) => *ContractId* i -> *Update* (*Optional* (*ContractId* t, t))

(Daml-LF >= 1.15) Fetch an interface and convert it to a specific template type. If conversion is succesful, this function returns the converted contract and its converted contract id. Otherwise, this function returns `None`.

Can also be used to fetch and convert an interface contract id into a contract and contract id of one of its requiring interfaces.

Example:

```
do
  fetchResult <- fetchFromInterface @MyTemplate ifaceCid
  case fetchResult of
    None -> abort "Failed to convert interface to appropriate template type"
    Some (tplCid, tpl) -> do
       ... do something with tpl and tplCid ...
```

_exerciseInterfaceGuard : a -> b -> c -> *Bool*

view : *HasInterfaceView* i v => i -> v

partyToText : *Party* -> *Text*

Convert the `Party` to `Text`, giving back what you passed to `getParty`. In most cases, you should use `show` instead. `show` wraps the party in `'ticks'` making it clear it was a `Party` originally.

partyFromText : *Text* -> *Optional Party*

Converts a `Text` to `Party`. It returns `None` if the provided text contains any forbidden characters. See Daml-LF spec for a specification on which characters are allowed in parties. Note that this function accepts text *without* single quotes.

This function does not check on whether the provided text corresponds to a party that "exists" on a given ledger: it merely converts the given `Text` to a `Party`. The only way to guarantee that a given `Party` exists on a given ledger is to involve it in a contract.

This function, together with `partyToText`, forms an isomorphism between valid party strings and parties. In other words, the following equations hold:

```
p. partyFromText (partyToText p) = Some p
txt p. partyFromText txt = Some p ==> partyToText p = txt
```

This function will crash at runtime if you compile Daml to Daml-LF < 1.2.

getParty : *Text* -> *Scenario Party*

Get the party with the given name. Party names must be non-empty and only contain alphanumeric charaters, space, – (dash) or _ (underscore).

coerceContractId : *ContractId* a -> *ContractId* b

Used to convert the type index of a `ContractId`, since they are just pointers. Note that subsequent fetches and exercises might fail if the template of the contract on the ledger doesn't match.

scenario : *Scenario* a -> *Scenario* a

Declare you are building a scenario.

curry : ((a, b) -> c) -> a -> b -> c

Turn a function that takes a pair into a function that takes two arguments.

uncurry : (a -> b -> c) -> (a, b) -> c
Turn a function that takes two arguments into a function that takes a pair.

(>>) : *Action* m => m a -> m b -> m b
Sequentially compose two actions, discarding any value produced by the first. This is like sequencing operators (such as the semicolon) in imperative languages.

ap : *Applicative* f => f (a -> b) -> f a -> f b
Synonym for `<*>`.

return : *Applicative* m => a -> m a
Inject a value into the monadic type. For example, for `Update` and a value of type `a`, `return` would give you an `Update a`.

join : *Action* m => m (m a) -> m a
Collapses nested actions into a single action.

identity : a -> a
The identity function.

guard : *ActionFail* m => *Bool* -> m ()

foldl : (b -> a -> b) -> b -> [a] -> b
This function is a left fold, which you can use to inspect/analyse/consume lists. `foldl f i xs` performs a left fold over the list `xs` using the function `f`, using the starting value `i`.
Examples:

```
>>> foldl (+) 0 [1,2,3]
6

>>> foldl (^) 10 [2,3]
1000000
```

Note that foldl works from left-to-right over the list arguments.

find : (a -> *Bool*) -> [a] -> *Optional* a
`find p xs` finds the first element of the list `xs` where the predicate `p` is true. There might not be such an element, which is why this function returns an `Optional a`.

length : [a] -> *Int*
Gives the length of the list.

any : (a -> *Bool*) -> [a] -> *Bool*
Are there any elements in the list where the predicate is true? `any p xs` is `True` if `p` holds for at least one element of `xs`.

all : (a -> *Bool*) -> [a] -> *Bool*
Is the predicate true for all of the elements in the list? `all p xs` is `True` if `p` holds for every element of `xs`.

or : [*Bool*] -> *Bool*
Is at least one of elements in a list of `Bool` true? `or bs` is `True` if at least one element of `bs` is `True`.

and : [*Bool*] -> *Bool*
Is every element in a list of Bool true? `and bs` is `True` if every element of `bs` is `True`.

elem : *Eq* a => a -> [a] -> *Bool*

Does this value exist in this list? `elem x xs` is `True` if `x` is an element of the list `xs`.

notElem : *Eq* a => a -> [a] -> *Bool*
Negation of `elem`: `elem x xs` is `True` if `x` is *not* an element of the list `xs`.

(<$>) : *Functor* f => (a -> b) -> f a -> f b
Synonym for `fmap`.

optional : b -> (a -> b) -> *Optional* a -> b
The `optional` function takes a default value, a function, and a `Optional` value. If the `Optional` value is `None`, the function returns the default value. Otherwise, it applies the function to the value inside the `Some` and returns the result.
Basic usage examples:

```
>>> optional False (> 2) (Some 3)
True
```

```
>>> optional False (> 2) None
False
```

```
>>> optional 0 (*2) (Some 5)
10
>>> optional 0 (*2) None
0
```

This example applies `show` to a `Optional Int`. If you have `Some n`, this shows the underlying `Int`, `n`. But if you have `None`, this returns the empty string instead of (for example) `None`:

```
>>> optional "" show (Some 5)
"5"
>>> optional "" show (None : Optional Int)
""
```

either : (a -> c) -> (b -> c) -> *Either* a b -> c
The `either` function provides case analysis for the `Either` type. If the value is `Left a`, it applies the first function to `a`; if it is `Right b`, it applies the second function to `b`.
Examples:
This example has two values of type `Either [Int] Int`, one using the `Left` constructor and another using the `Right` constructor. Then it applies `either` the `length` function (if it has a `[Int]`) or the "times-two" function (if it has an `Int`):

```
>>> let s = Left [1,2,3] : Either [Int] Int in either length (*2) s
3
>>> let n = Right 3 : Either [Int] Int in either length (*2) n
6
```

concat : [[a]] -> [a]
Take a list of lists and concatenate those lists into one list.

(++) : [a] -> [a] -> [a]
Concatenate two lists.

flip : (a -> b -> c) -> b -> a -> c
Flip the order of the arguments of a two argument function.

reverse : [a] -> [a]
Reverse a list.

mapA : *Applicative* m => (a -> m b) -> [a] -> m [b]
Apply an applicative function to each element of a list.

forA : *Applicative* m => [a] -> (a -> m b) -> m [b]
`forA` is `mapA` with its arguments flipped.

sequence : *Applicative* m => [m a] -> m [a]
Perform a list of actions in sequence and collect the results.

(=<<) : *Action* m => (a -> m b) -> m a -> m b
=<< is >>= with its arguments flipped.

concatMap : (a -> [b]) -> [a] -> [b]
Map a function over each element of a list, and concatenate all the results.

replicate : *Int* -> a -> [a]
`replicate i x` gives the list `[x, x, x, ..., x]` with `i` copies of `x`.

take : *Int* -> [a] -> [a]
Take the first `n` elements of a list.

drop : *Int* -> [a] -> [a]
Drop the first `n` elements of a list.

splitAt : *Int* -> [a] -> ([a], [a])
Split a list at a given index.

takeWhile : (a -> *Bool*) -> [a] -> [a]
Take elements from a list while the predicate holds.

dropWhile : (a -> *Bool*) -> [a] -> [a]
Drop elements from a list while the predicate holds.

span : (a -> *Bool*) -> [a] -> ([a], [a])
`span p xs` is equivalent to `(takeWhile p xs, dropWhile p xs)`.

partition : (a -> *Bool*) -> [a] -> ([a], [a])
The `partition` function takes a predicate, a list and returns the pair of lists of elements which do and do not satisfy the predicate, respectively; i.e.,
> partition p xs == (filter p xs, filter (not . p) xs)

```
>>> partition (<0) [1, -2, -3, 4, -5, 6]
([-2, -3, -5], [1, 4, 6])
```

break : (a -> *Bool*) -> [a] -> ([a], [a])
Break a list into two, just before the first element where the predicate holds. `break p xs` is equivalent to `span (not . p) xs`.

lookup : *Eq* a => a -> [(a, b)] -> *Optional* b
Look up the first element with a matching key.

enumerate : (*Enum* a, *Bounded* a) => [a]
Generate a list containing all values of a given enumeration.

zip : [a] -> [b] -> [(a, b)]
`zip` takes two lists and returns a list of corresponding pairs. If one list is shorter, the excess elements of the longer list are discarded.

zip3 : [a] -> [b] -> [c] -> [(a, b, c)]
`zip3` takes three lists and returns a list of triples, analogous to `zip`.

zipWith : (a -> b -> c) -> [a] -> [b] -> [c]
`zipWith` takes a function and two lists. It generalises `zip` by combining elements using the function, instead of forming pairs. If one list is shorter, the excess elements of the longer list are discarded.

zipWith3 : (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
`zipWith3` generalises `zip3` by combining elements using the function, instead of forming triples.

unzip : [(a, b)] -> ([a], [b])
Turn a list of pairs into a pair of lists.

unzip3 : [(a, b, c)] -> ([a], [b], [c])
Turn a list of triples into a triple of lists.

traceRaw : *Text* -> a -> a
`traceRaw msg a` prints `msg` and returns `a`, for debugging purposes.
Note that on some ledgers, those messages are not displayed at the default log level. For Sandbox, you can use `--log-level=debug` to include them.

trace : *Show* b => b -> a -> a
`trace b a` prints `b` and returns `a`, for debugging purposes.
Note that on some ledgers, those messages are not displayed at the default log level. For Sandbox, you can use `--log-level=debug` to include them.

traceId : *Show* b => b -> b
`traceId a` prints `a` and returns `a`, for debugging purposes.
Note that on some ledgers, those messages are not displayed at the default log level. For Sandbox, you can use `--log-level=debug` to include them.

debug : (*Show* b, *Action* m) => b -> m ()
`debug x` prints `x` for debugging purposes.
Note that on some ledgers, those messages are not displayed at the default log level. For Sandbox, you can use `--log-level=debug` to include them.

debugRaw : *Action* m => *Text* -> m ()
`debugRaw msg` prints `msg` for debugging purposes.
Note that on some ledgers, those messages are not displayed at the default log level. For Sandbox, you can use `--log-level=debug` to include them.

fst : (a, b) -> a
Return the first element of a tuple.

snd : (a, b) -> b
Return the second element of a tuple.

truncate : *Numeric* n -> *Int*
`truncate x` rounds `x` toward zero.

intToNumeric : *Int* -> *Numeric* n
Convert an `Int` to a `Numeric`.

intToDecimal : *Int* -> *Decimal*
Convert an `Int` to a `Decimal`.

roundBankers : *Int* -> *Numeric* n -> *Numeric* n
Bankers' Rounding: `roundBankers dp x` rounds `x` to `dp` decimal places, where a `.5` is rounded to the nearest even digit.

roundCommercial : *NumericScale* n => *Int* -> *Numeric* n -> *Numeric* n
Commercial Rounding: `roundCommercial dp x` rounds `x` to `dp` decimal places, where a `.5` is rounded away from zero.

round : *Numeric* n -> *Int*
Round a `Decimal` to the nearest integer, where a `.5` is rounded away from zero.

floor : *Numeric* n -> *Int*
Round a `Decimal` down to the nearest integer.

ceiling : *Numeric* n -> *Int*
Round a `Decimal` up to the nearest integer.

null : [a] -> *Bool*
Is the list empty? `null xs` is true if `xs` is the empty list.

filter : (a -> *Bool*) -> [a] -> [a]
Filters the list using the function: keep only the elements where the predicate holds.

sum : *Additive* a => [a] -> a
Add together all the elements in the list.

product : *Multiplicative* a => [a] -> a
Multiply all the elements in the list together.

undefined : a
A convenience function that can be used to mark something not implemented. Always throws an error with "Not implemented."

stakeholder : (*HasSignatory* t, *HasObserver* t) => t -> [*Party*]
The stakeholders of a contract: its signatories and observers.

maintainer : *HasMaintainer* t k => k -> [*Party*]
The list of maintainers of a contract key.

exerciseByKey : *HasExerciseByKey* t k c r => k -> c -> *Update* r
Exercise a choice on the contract associated with the given key.
You must pass the `t` using an explicit type application. For instance, if you want to exercise a choice `Withdraw` on a contract of template `Account` given by its key `k`, you must call `exerciseByKey @Account k Withdraw`.

createAndExercise : (*HasCreate* t, *HasExercise* t c r) => t -> c -> *Update* r
Create a contract and exercise the choice on the newly created contract.

templateTypeRep : *HasTemplateTypeRep* t => *TemplateTypeRep*
Generate a unique textual representation of the template id.

toAnyTemplate : *HasToAnyTemplate* t => t -> *AnyTemplate*
Wrap the template in `AnyTemplate`.
Only available for Daml-LF 1.7 or later.

fromAnyTemplate : *HasFromAnyTemplate* t => *AnyTemplate* -> *Optional* t
Extract the underlying template from `AnyTemplate` if the type matches or return `None`.
Only available for Daml-LF 1.7 or later.

toAnyChoice : (*HasTemplateTypeRep* t, *HasToAnyChoice* t c r) => c -> *AnyChoice*
Wrap a choice in `AnyChoice`.
You must pass the template type `t` using an explicit type application. For example `toAnyChoice @Account Withdraw`.

Only available for Daml-LF 1.7 or later.

fromAnyChoice : (*HasTemplateTypeRep* t, *HasFromAnyChoice* t c r) => *AnyChoice* -> *Optional* c
Extract the underlying choice from `AnyChoice` if the template and choice types match, or return `None`.
You must pass the template type `t` using an explicit type application. For example `fromAnyChoice @Account choice`.
Only available for Daml-LF 1.7 or later.

toAnyContractKey : (*HasTemplateTypeRep* t, *HasToAnyContractKey* t k) => k -> *AnyContractKey*
Wrap a contract key in `AnyContractKey`.
You must pass the template type `t` using an explicit type application. For example `toAnyContractKey @Proposal k`.
Only available for Daml-LF 1.7 or later.

fromAnyContractKey : (*HasTemplateTypeRep* t, *HasFromAnyContractKey* t k) => *AnyContractKey* -> *Optional* k
Extract the underlying key from `AnyContractKey` if the template and choice types match, or return `None`.
You must pass the template type `t` using an explicit type application. For example `fromAnyContractKey @Proposal k`.
Only available for Daml-LF 1.7 or later.

visibleByKey : *HasLookupByKey* t k => k -> *Update Bool*
True if contract exists, submitter is a stakeholder, and all maintainers authorize. False if contract does not exist and all maintainers authorize. Fails otherwise.

otherwise : *Bool*
Used as an alternative in conditions.

map : (a -> b) -> [a] -> [b]
`map f xs` applies the function `f` to all elements of the list `xs` and returns the list of results (in the same order as `xs`).

foldr : (a -> b -> b) -> b -> [a] -> b
This function is a right fold, which you can use to manipulate lists. `foldr f i xs` performs a right fold over the list `xs` using the function `f`, using the starting value `i`.
Note that foldr works from right-to-left over the list elements.

(.) : (b -> c) -> (a -> b) -> a -> c
Composes two functions, i.e., `(f . g) x = f (g x)`.

const : a -> b -> a
`const x` is a unary function which evaluates to `x` for all inputs.

```
>>> const 42 "hello"
42
```

```
>>> map (const 42) [0..3]
[42,42,42,42]
```

($) : (a -> b) -> a -> b
Take a function from `a` to `b` and a value of type `a`, and apply the function to the value of type `a`, returning a value of type `b`. This function has a very low precedence, which is why you might want to use it instead of regular function application.

(&&) : *Bool* -> *Bool* -> *Bool*

Boolean "and". This function has short-circuiting semantics, i.e., when both arguments are present and the first arguments evaluates to 'False', the second argument is not evaluated at all.

(||) : *Bool* -> *Bool* -> *Bool*
Boolean "or". This function has short-circuiting semantics, i.e., when both arguments are present and the first arguments evaluates to 'True', the second argument is not evaluated at all.

not : *Bool* -> *Bool*
Boolean "not"

error : *Text* -> a
Throws a `GeneralError` exception.

subtract : *Additive* a => a -> a -> a
`subtract x y` is equivalent to `y - x`.
This is useful for partial application, e.g., in `subtract 1` since `(- 1)` is interpreted as the number `-1` and not a function that subtracts `1` from its argument.

(%) : *Int* -> *Int* -> *Int*
`x % y` calculates the remainder of `x` by `y`

shows : *Show* a => a -> *ShowS*

showParen : *Bool* -> *ShowS* -> *ShowS*
Utility function that surrounds the inner show function with parentheses when the 'Bool' parameter is 'True'.

showString : *Text* -> *ShowS*
Utility function converting a 'String' to a show function that simply prepends the string unchanged.

showSpace : *ShowS*
Prepends a single space to the front of the string.

showCommaSpace : *ShowS*
Prepends a comma and a single space to the front of the string.

1.43.3.2 DA.Action

Action

Functions

when : *Applicative* f => *Bool* -> f () -> f ()
Conditional execution of `Action` expressions. For example,

```
when final (archive contractId)
```

will archive the contract `contractId` if the Boolean value `final` is `True`, and otherwise do nothing.
This function has short-circuiting semantics, i.e., when both arguments are present and the first arguments evaluates to `False`, the second argument is not evaluated at all.

unless : *Applicative* f => *Bool* -> f () -> f ()
The reverse of `when`.
This function has short-circuiting semantics, i.e., when both arguments are present and the first arguments evaluates to `True`, the second argument is not evaluated at all.

foldrA : *Action* m => (a -> b -> m b) -> b -> [a] -> m b
The `foldrA` is analogous to `foldr`, except that its result is encapsulated in an action. Note that `foldrA` works from right-to-left over the list arguments.

foldr1A : *Action* m => (a -> a -> m a) -> [a] -> m a
`foldr1A` is like `foldrA` but raises an error when presented with an empty list argument.

foldlA : *Action* m => (b -> a -> m b) -> b -> [a] -> m b
`foldlA` is analogous to `foldl`, except that its result is encapsulated in an action. Note that `foldlA` works from left-to-right over the list arguments.

foldl1A : *Action* m => (a -> a -> m a) -> [a] -> m a
The `foldl1A` is like `foldlA` but raises an errors when presented with an empty list argument.

filterA : *Applicative* m => (a -> m *Bool*) -> [a] -> m [a]
Filters the list using the applicative function: keeps only the elements where the predicate holds. Example: given a collection of Iou contract IDs one can find only the GBPs.

```
filterA (fmap (\iou -> iou.currency == "GBP") . fetch) iouCids
```

replicateA : *Applicative* m => *Int* -> m a -> m [a]
`replicateA n act` performs the action `n` times, gathering the results.

replicateA_ : *Applicative* m => *Int* -> m a -> m ()
Like `replicateA`, but discards the result.

(>=>) : *Action* m => (a -> m b) -> (b -> m c) -> a -> m c
Left-to-right composition of Kleisli arrows.

(<=<) : *Action* m => (b -> m c) -> (a -> m b) -> a -> m c
Right-to-left composition of Kleisli arrows. @('>=>')@, with the arguments flipped.

1.43.3.3 DA.Action.State

DA.Action.State

Data Types

data *State* s a

A value of type `State s a` represents a computation that has access to a state variable of type `s` and produces a value of type `a`.

> > > runState (modify (+1)) 0 > > > ((), 1)

> > > evalState (modify (+1)) 0 > > > ()

> > > execState (modify (+1)) 0 > > > 1

> > > runState (do x <- get; modify (+1); pure x) 0 > > > (0, 1)

> > > runState (put 1) 0 > > > ((), 1)

> > > runState (modify (+1)) 0 > > > ((), 1)

Note that values of type `State s a` are not serializable.

State

Field	Type	Description
runState	s -> (a, s)	

instance *ActionState* s (*State* s)

instance *Action* (*State* s)

instance *Applicative* (*State* s)

instance *Functor* (*State* s)

Functions

evalState : *State* s a -> s -> a
Special case of `runState` that does not return the final state.

execState : *State* s a -> s -> s
Special case of `runState` that does only retun the final state.

1.43.3.4 DA.Action.State.Class

DA.Action.State.Class

Typeclasses

class *ActionState* s m **where**

Action `m` has a state variable of type `s`.

Rules:

```
get *> ma = ma
ma <* get = ma
put a >>= get = put a $> a
put a *> put b = put b
(,) <$> get <*> get = get <&> \a -> (a, a)
```

Informally, these rules mean it behaves like an ordinary assignable variable: it doesn't magically change value by looking at it, if you put a value there that's always the value you'll get if you read it, assigning a value but never reading that value has no effect, and so on.

get : m s
Fetch the current value of the state variable.

put : s -> m ()
Set the value of the state variable.

modify : (s -> s) -> m ()
Modify the state variable with the given function.

default modify

: *Action* m => (s -> s) -> m ()

instance *ActionState* s (*State* s)

1.43.3.5 DA.Assert

Functions

assertEq : (*CanAssert* m, *Show* a, *Eq* a) => a -> a -> m ()
Check two values for equality. If they're not equal, fail with a message.

(===) : (*CanAssert* m, *Show* a, *Eq* a) => a -> a -> m ()
Infix version of `assertEq`.

assertNotEq : (*CanAssert* m, *Show* a, *Eq* a) => a -> a -> m ()
Check two values for inequality. If they're equal, fail with a message.

(=/=) : (*CanAssert* m, *Show* a, *Eq* a) => a -> a -> m ()
Infix version of `assertNotEq`.

assertAfterMsg : (*CanAssert* m, *HasTime* m) => *Text* -> *Time* -> m ()
Check whether the given time is in the future. If it's not, abort with a message.

assertBeforeMsg : (*CanAssert* m, *HasTime* m) => *Text* -> *Time* -> m ()
Check whether the given time is in the past. If it's not, abort with a message.

1.43.3.6 DA.Bifunctor

Typeclasses

class *Bifunctor* p **where**

A bifunctor is a type constructor that takes two type arguments and is a functor in *both* arguments. That is, unlike with `Functor`, a type constructor such as `Either` does not need to be partially applied for a `Bifunctor` instance, and the methods in this class permit mapping functions over the Left value or the `Right` value, or both at the same time.

It is a bifunctor where both the first and second arguments are covariant.

You can define a `Bifunctor` by either defining bimap or by defining both first and second.

If you supply bimap, you should ensure that:

```
`bimap identity identity` ≡ `identity`
```

If you supply first and second, ensure:

```
first identity ≡ identity
second identity ≡ identity
```

If you supply both, you should also ensure:

```
bimap f g ≡ first f . second g
```

By parametricity, these will ensure that:

```
bimap  (f . g) (h . i) ≡ bimap f h . bimap g i
first  (f . g) ≡ first  f . first  g
second (f . g) ≡ second f . second g
```

bimap : (a -> b) -> (c -> d) -> p a c -> p b d

Map over both arguments at the same time.

```
bimap f g ≡ first f . second g
```

Examples:

```
>>> bimap not (+1) (True, 3)
(False,4)

>>> bimap not (+1) (Left True)
Left False

>>> bimap not (+1) (Right 3)
Right 4
```

first : (a -> b) -> p a c -> p b c

Map covariantly over the first argument.

```
first f ≡ bimap f identity
```

Examples:

```
>>> first not (True, 3)
(False,3)

>>> first not (Left True : Either Bool Int)
Left False
```

second : (b -> c) -> p a b -> p a c

Map covariantly over the second argument.

```
second ≡ bimap identity
```

Examples:

```
>>> second (+1) (True, 3)
(True,4)

>>> second (+1) (Right 3 : Either Bool Int)
Right 4
```

instance *Bifunctor Either*

instance *Bifunctor* ()

instance *Bifunctor* x1

instance *Bifunctor* (x1, x2)

instance *Bifunctor* (x1, x2, x3)

instance *Bifunctor* (x1, x2, x3, x4)

instance *Bifunctor* (x1, x2, x3, x4, x5)

1.43.3.7 DA.BigNumeric

This module exposes operations for working with the `BigNumeric` type.

Functions

scale : *BigNumeric* -> *Int*

Calculate the scale of a `BigNumeric` number. The `BigNumeric` number is represented as `n * 10^-s` where `n` is an integer with no trailing zeros, and `s` is the scale.
Thus, the scale represents the number of nonzero digits after the decimal point. Note that the scale can be negative if the `BigNumeric` represents an integer with trailing zeros. In that case, it represents the number of trailing zeros (negated).
The scale ranges between 2^15 and -2^15 + 1. The scale of `0` is `0` by convention.

```
>>> scale 1.1
1
```

```
>>> scale (shiftLeft (2^14) 1.0)
-2^14
```

precision : *BigNumeric* -> *Int*

Calculate the precision of a `BigNumeric` number. The `BigNumeric` number is represented as `n * 10^-s` where `n` is an integer with no trailing zeros, and `s` is the scale. The precision is the number of digits in `n`.
Thus, the precision represents the number of significant digits in the `BigNumeric`.
The precision ranges between 0 and 2^16 - 1.

```
>>> precision 1.10
2
```

div : *Int* -> *RoundingMode* -> *BigNumeric* -> *BigNumeric* -> *BigNumeric*

Calculate a division of `BigNumeric` numbers. The value of `div n r a b` is the division of `a` by `b`, rounded to `n` decimal places (i.e. scale), according to the rounding mode `r`.
This will fail when dividing by `0`, and when using the `RoundingUnnecessary` mode for a number that cannot be represented exactly with at most `n` decimal places.

round : *Int* -> *RoundingMode* -> *BigNumeric* -> *BigNumeric*

Round a `BigNumeric` number. The value of `round n r a` is the value of `a` rounded to `n` decimal places (i.e. scale), according to the rounding mode `r`.
This will fail when using the `RoundingUnnecessary` mode for a number that cannot be represented exactly with at most `n` decimal places.

shiftRight : *Int* -> *BigNumeric* -> *BigNumeric*

Shift a `BigNumeric` number to the right by a power of 10. The value `shiftRight n a` is the value of `a` times `10^(-n)`.
This will fail if the resulting `BigNumeric` is out of bounds.

```
>>> shiftRight 2 32.0
0.32
```

shiftLeft : *Int* -> *BigNumeric* -> *BigNumeric*

Shift a `BigNumeric` number to the left by a power of 10. The value `shiftLeft n a` is the value of `a` times `10^n`.
This will fail if the resulting `BigNumeric` is out of bounds.

```
>>> shiftLeft 2 32.0
3200.0
```

roundToNumeric : *NumericScale* n => *RoundingMode* -> *BigNumeric* -> *Numeric* n

Round a `BigNumeric` and cast it to a `Numeric`. This function uses the scale of the resulting numeric to determine the scale of the rounding.
This will fail when using the `RoundingUnnecessary` mode if the `BigNumeric` cannot be represented exactly in the requested `Numeric n`.

```
>>> (roundToNumeric RoundingHalfUp 1.23456789 : Numeric 5)
1.23457
```

1.43.3.8 DA.Date

Data Types

data *DayOfWeek*

Monday

Tuesday

Wednesday

Thursday

Friday

Saturday

Sunday

instance *Eq DayOfWeek*

instance *Ord DayOfWeek*

instance *Bounded DayOfWeek*

instance *Enum DayOfWeek*

instance *Show DayOfWeek*

data *Month*

The `Month` type represents a month in the Gregorian calendar.

Note that, while `Month` has an `Enum` instance, the `toEnum` and `fromEnum` functions start counting at 0, i.e. `toEnum 1 :: Month` is `Feb`.

Jan

Feb

Mar

Apr

May

Jun

Jul

Aug

Sep

Oct

Nov

Dec

instance *Eq Month*

instance *Ord Month*

instance *Bounded Month*

instance *Enum Month*

instance *Show Month*

Functions

addDays : *Date* -> *Int* -> *Date*
Add the given number of days to a date.

subtractDays : *Date* -> *Int* -> *Date*
Subtract the given number of days from a date.
`subtractDays d r` is equivalent to `addDays d (- r)`.

subDate : *Date* -> *Date* -> *Int*
Returns the number of days between the two given dates.

dayOfWeek : *Date* -> *DayOfWeek*
Returns the day of week for the given date.

fromGregorian : (*Int*, *Month*, *Int*) -> *Date*
Constructs a `Date` from the triplet `(year, month, days)`.

toGregorian : *Date* -> (*Int*, *Month*, *Int*)
Turn `Date` value into a `(year, month, day)` triple, according to the Gregorian calendar.

date : *Int* -> *Month* -> *Int* -> *Date*
Given the three values (year, month, day), constructs a `Date` value. `date y m d` turns the year `y`, month `m`, and day `d` into a `Date` value. Raises an error if `d` is outside the range `1 .. monthDayCount y m`.

isLeapYear : *Int* -> *Bool*
Returns `True` if the given year is a leap year.

fromMonth : *Month* -> *Int*
Get the number corresponding to given month. For example, `Jan` corresponds to `1`, `Feb` corresponds to `2`, and so on.

monthDayCount : *Int* -> *Month* -> *Int*
Get number of days in the given month in the given year, according to Gregorian calendar. This does not take historical calendar changes into account (for example, the moves from Julian to Gregorian calendar), but does count leap years.

datetime : *Int* -> *Month* -> *Int* -> *Int* -> *Int* -> *Int* -> *Time*
Constructs an instant using `year`, `month`, `day`, `hours`, `minutes`, `seconds`.

toDateUTC : *Time* -> *Date*
Extracts UTC date from UTC time.
This function will truncate Time to Date, but in many cases it will not return the date you really want. The reason for this is that usually the source of Time would be getTime, and getTime returns UTC, and most likely the date you want is something local to a location or an exchange. Consequently the date retrieved this way would be yesterday if retrieved when the market opens in say Singapore.

passToDate : *Date* -> *Scenario Time*
Within a `scenario`, pass the simulated scenario to given date.

1.43.3.9 DA.Either

The `Either` type represents values with two possibilities.

It is sometimes used to represent a value which is either correct or an error. By convention, the `Left` constructor is used to hold an error value and the `Right` constructor is used to hold a correct value (mnemonic: "right" also means correct).

Functions

lefts : [*Either* a b] -> [a]
Extracts all the `Left` elements from a list.

rights : [*Either* a b] -> [b]
Extracts all the `Right` elements from a list.

partitionEithers : [*Either* a b] -> ([a], [b])
Partitions a list of `Either` into two lists, the `Left` and `Right` elements respectively. Order is maintained.

isLeft : *Either* a b -> *Bool*
Return `True` if the given value is a `Left`-value, `False` otherwise.

isRight : *Either* a b -> *Bool*
Return `True` if the given value is a `Right`-value, `False` otherwise.

fromLeft : a -> *Either* a b -> a
Return the contents of a `Left`-value, or a default value in case of a `Right`-value.

fromRight : b -> *Either* a b -> b
Return the contents of a `Right`-value, or a default value in case of a `Left`-value.

optionalToEither : a -> *Optional* b -> *Either* a b
Convert a `Optional` value to an `Either` value, using the supplied parameter as the `Left` value if the `Optional` is `None`.

eitherToOptional : *Either* a b -> *Optional* b
Convert an `Either` value to a `Optional`, dropping any value in `Left`.

maybeToEither : a -> *Optional* b -> *Either* a b

eitherToMaybe : *Either* a b -> *Optional* b

1.43.3.10 DA.Exception

Exception handling in Daml.

Typeclasses

class *HasThrow* e **where**

Part of the `Exception` constraint.

throwPure : e -> t
Throw exception in a pure context.

instance *HasThrow ArithmeticError*

instance *HasThrow AssertionFailed*

instance *HasThrow GeneralError*

instance *HasThrow PreconditionFailed*

class *HasMessage* e **where**

Part of the `Exception` constraint.

message : e -> *Text*
Get the error message associated with an exception.

instance *HasMessage AnyException*

instance *HasMessage ArithmeticError*

instance *HasMessage AssertionFailed*

instance *HasMessage GeneralError*

instance *HasMessage PreconditionFailed*

class *HasToAnyException* e **where**

Part of the `Exception` constraint.

toAnyException : e -> *AnyException*
Convert an exception type to AnyException.

instance *HasToAnyException AnyException*

instance *HasToAnyException ArithmeticError*

instance *HasToAnyException AssertionFailed*

instance *HasToAnyException GeneralError*

instance *HasToAnyException PreconditionFailed*

class *HasFromAnyException* e **where**

Part of the `Exception` constraint.

fromAnyException : *AnyException* -> *Optional* e
Convert an AnyException back to the underlying exception type, if possible.

instance *HasFromAnyException AnyException*

instance *HasFromAnyException ArithmeticError*

instance *HasFromAnyException AssertionFailed*

instance *HasFromAnyException GeneralError*

instance *HasFromAnyException PreconditionFailed*

class *Action* m => *ActionThrow* m **where**

Action type in which `throw` is supported.

throw : *Exception* e => e -> m t

instance *ActionThrow Scenario*

instance *ActionThrow Update*

class *ActionThrow* m => *ActionCatch* m **where**

Action type in which `try ... catch ...` is supported.

_tryCatch : (() -> m t) -> (*AnyException* -> *Optional* (m t)) -> m t
Handle an exception. Use the `try ... catch ...` syntax instead of calling this method directly.

instance *ActionCatch Update*

Data Types

type ***Exception*** **e** = (*HasThrow* e, *HasMessage* e, *HasToAnyException* e, *HasFromAnyException* e)
Exception typeclass. This should not be implemented directly, instead, use the `exception` syntax.

data *ArithmeticError*

Exception raised by an arithmetic operation, such as divide-by-zero or overflow.

ArithmeticError

Field	Type	Description
message	*Text*	

data *AssertionFailed*

Exception raised by assert functions in DA.Assert

AssertionFailed

Field	Type	Description
message	*Text*	

data *GeneralError*

Exception raised by `error`.

GeneralError

Field	Type	Description
message	*Text*	

data *PreconditionFailed*

Exception raised when a contract is invalid, i.e. fails the ensure clause.

PreconditionFailed

Field	Type	Description
message	*Text*	

1.43.3.11 DA.Foldable

Class of data structures that can be folded to a summary value. It's a good idea to import this module qualified to avoid clashes with functions defined in `Prelude`. Ie.:

```
import DA.Foldable qualified as F
```

Typeclasses

class *Foldable* t **where**

Class of data structures that can be folded to a summary value.

fold : *Monoid* m => t m -> m
Combine the elements of a structure using a monoid.

foldMap : *Monoid* m => (a -> m) -> t a -> m
Combine the elements of a structure using a monoid.

foldr : (a -> b -> b) -> b -> t a -> b
Right-associative fold of a structure.

foldl : (b -> a -> b) -> b -> t a -> b
Left-associative fold of a structure.

foldr1 : (a -> a -> a) -> t a -> a
A variant of foldr that has no base case, and thus should only be applied to non-empty structures.

foldl1 : (a -> a -> a) -> t a -> a
A variant of foldl that has no base case, and thus should only be applied to non-empty structures.

toList : t a -> [a]
List of elements of a structure, from left to right.

null : t a -> *Bool*
Test whether the structure is empty. The default implementation is optimized for structures that are similar to cons-lists, because there is no general way to do better.

length : t a -> *Int*
Returns the size/length of a finite structure as an `Int`. The default implementation is optimized for structures that are similar to cons-lists, because there is no general way to do better.

elem : *Eq* a => a -> t a -> *Bool*
Does the element occur in the structure?

sum : *Additive* a => t a -> a
The sum function computes the sum of the numbers of a structure.

product : *Multiplicative* a => t a -> a
The product function computes the product of the numbers of a structure.

minimum : *Ord* a => t a -> a
The least element of a non-empty structure.

maximum : *Ord* a => t a -> a
The largest element of a non-empty structure.

instance *Ord* k => *Foldable* (*Map* k)

instance *Foldable TextMap*

instance *Foldable Optional*

instance *Foldable NonEmpty*

instance *Foldable Set*

instance *Foldable* (*Validation* err)

instance *Foldable* (*Either* a)

instance *Foldable ([])*

instance *Foldable* a

Functions

mapA_ : (*Foldable* t, *Applicative* f) => (a -> f b) -> t a -> f ()
Map each element of a structure to an action, evaluate these actions from left to right, and ignore the results. For a version that doesn't ignore the results see 'DA.Traversable.mapA'.

forA_ : (*Foldable* t, *Applicative* f) => t a -> (a -> f b) -> f ()
'for_' is 'mapA_' with its arguments flipped. For a version that doesn't ignore the results see 'DA.Traversable.forA'.

forM_ : (*Foldable* t, *Applicative* f) => t a -> (a -> f b) -> f ()

sequence_ : (*Foldable* t, *Action* m) => t (m a) -> m ()
Evaluate each action in the structure from left to right, and ignore the results. For a version that doesn't ignore the results see 'DA.Traversable.sequence'.

concat : *Foldable* t => t [a] -> [a]
The concatenation of all the elements of a container of lists.

and : *Foldable* t => t *Bool* -> *Bool*
`and` returns the conjunction of a container of Bools. For the result to be `True`, the container must be finite; `False`, however, results from a `False` value finitely far from the left end.

or : *Foldable* t => t *Bool* -> *Bool*
`or` returns the disjunction of a container of Bools. For the result to be `False`, the container must be finite; `True`, however, results from a `True` value finitely far from the left end.

any : *Foldable* t => (a -> *Bool*) -> t a -> *Bool*
Determines whether any element of the structure satisfies the predicate.

all : *Foldable* t => (a -> *Bool*) -> t a -> *Bool*
Determines whether all elements of the structure satisfy the predicate.

1.43.3.12 DA.Functor

The `Functor` class is used for types that can be mapped over.

Functions

($>) : *Functor* f => f a -> b -> f b
Replace all locations in the input (on the left) with the given value (on the right).

(<&>) : *Functor* f => f a -> (a -> b) -> f b
Map a function over a functor. Given a value `as` and a function `f`, `as <&> f` is `f <$> as`. That is, `<&>` is like `<$>` but the arguments are in reverse order.

void : *Functor* f => f a -> f ()
Replace all the locations in the input with `()`.

1.43.3.13 DA.Internal.Interface.AnyView

Typeclasses

class *HasFromAnyView* i v **where**

Functions

fromAnyView : (*HasTemplateTypeRep* i, *HasFromAnyView* i v) => *AnyView* -> *Optional* v

1.43.3.14 DA.Internal.Interface.AnyView.Types

Data Types

data *AnyView*

Existential contract key type that can wrap an arbitrary contract key.

AnyView

Field	Type	Description
getAnyView	Any	
getAnyViewInterfaceTypeRep	*InterfaceTypeRep*	

data *InterfaceTypeRep*

InterfaceTypeRep

Field	Type	Description
getInterfaceTypeRep	TypeRep	

instance *Eq InterfaceTypeRep*

instance *Ord InterfaceTypeRep*

1.43.3.15 DA.List

List

Functions

sort : *Ord* a => [a] -> [a]
The `sort` function implements a stable sorting algorithm. It is a special case of `sortBy`, which allows the programmer to supply their own comparison function.
Elements are arranged from lowest to highest, keeping duplicates in the order they appeared in the input (a stable sort).

sortBy : (a -> a -> *Ordering*) -> [a] -> [a]
The `sortBy` function is the non-overloaded version of `sort`.

minimumBy : (a -> a -> *Ordering*) -> [a] -> a
`minimumBy f xs` returns the first element `x` of `xs` for which `f x y` is either `LT` or `EQ` for all other `y` in `xs`. `xs` must be non-empty.

maximumBy : (a -> a -> *Ordering*) -> [a] -> a
`maximumBy f xs` returns the first element `x` of `xs` for which `f x y` is either `GT` or `EQ` for all other `y` in `xs`. `xs` must be non-empty.

sortOn : *Ord* k => (a -> k) -> [a] -> [a]
Sort a list by comparing the results of a key function applied to each element. `sortOn f` is equivalent to `sortBy (comparing f)`, but has the performance advantage of only evaluating `f` once for each element in the input list. This is sometimes called the decorate-sort-undecorate paradigm.
Elements are arranged from from lowest to highest, keeping duplicates in the order they appeared in the input.

minimumOn : *Ord* k => (a -> k) -> [a] -> a
`minimumOn f xs` returns the first element `x` of `xs` for which `f x` is smaller than or equal to any other `f y` for `y` in `xs`. `xs` must be non-empty.

maximumOn : *Ord* k => (a -> k) -> [a] -> a
`maximumOn f xs` returns the first element `x` of `xs` for which `f x` is greater than or equal to any other `f y` for `y` in `xs`. `xs` must be non-empty.

mergeBy : (a -> a -> *Ordering*) -> [a] -> [a] -> [a]
Merge two sorted lists using into a single, sorted whole, allowing the programmer to specify the comparison function.

combinePairs : (a -> a -> a) -> [a] -> [a]
Combine elements pairwise by means of a programmer supplied function from two list inputs into a single list.

foldBalanced1 : (a -> a -> a) -> [a] -> a
Fold a non-empty list in a balanced way. Balanced means that each element has approximately the same depth in the operator tree. Approximately the same depth means that the difference between maximum and minimum depth is at most 1. The accumulation operation must be associative and commutative in order to get the same result as `foldl1` or `foldr1`.

group : *Eq* a => [a] -> [[a]]
The 'group' function groups equal elements into sublists such that the concatenation of the result is equal to the argument.

groupBy : (a -> a -> *Bool*) -> [a] -> [[a]]
The 'groupBy' function is the non-overloaded version of 'group'.

groupOn : *Eq* k => (a -> k) -> [a] -> [[a]]

Similar to 'group', except that the equality is done on an extracted value.

dedup : *Ord* a => [a] -> [a]
`dedup l` removes duplicate elements from a list. In particular, it keeps only the first occurrence of each element. It is a special case of `dedupBy`, which allows the programmer to supply their own equality test. `dedup` is called `nub` in Haskell.

dedupBy : (a -> a -> *Ordering*) -> [a] -> [a]
A version of `dedup` with a custom predicate.

dedupOn : *Ord* k => (a -> k) -> [a] -> [a]
A version of `dedup` where deduplication is done after applyng function. Example use: `dedupOn (.employeeNo) employees`

dedupSort : *Ord* a => [a] -> [a]
The `dedupSort` function sorts and removes duplicate elements from a list. In particular, it keeps only the first occurrence of each element.

dedupSortBy : (a -> a -> *Ordering*) -> [a] -> [a]
A version of `dedupSort` with a custom predicate.

unique : *Ord* a => [a] -> *Bool*
Returns True if and only if there are no duplicate elements in the given list.

uniqueBy : (a -> a -> *Ordering*) -> [a] -> *Bool*
A version of `unique` with a custom predicate.

uniqueOn : *Ord* k => (a -> k) -> [a] -> *Bool*
Returns True if and only if there are no duplicate elements in the given list after applyng function. Example use: `assert $ uniqueOn (.employeeNo) employees`

replace : *Eq* a => [a] -> [a] -> [a] -> [a]
Given a list and a replacement list, replaces each occurance of the search list with the replacement list in the operation list.

dropPrefix : *Eq* a => [a] -> [a] -> [a]
Drops the given prefix from a list. It returns the original sequence if the sequence doesn't start with the given prefix.

dropSuffix : *Eq* a => [a] -> [a] -> [a]
Drops the given suffix from a list. It returns the original sequence if the sequence doesn't end with the given suffix.

stripPrefix : *Eq* a => [a] -> [a] -> *Optional* [a]
The `stripPrefix` function drops the given prefix from a list. It returns `None` if the list did not start with the prefix given, or `Some` the list after the prefix, if it does.

stripSuffix : *Eq* a => [a] -> [a] -> *Optional* [a]
Return the prefix of the second list if its suffix matches the entire first list.

stripInfix : *Eq* a => [a] -> [a] -> *Optional* ([a], [a])
Return the string before and after the search string or `None` if the search string is not found.

```
>>> stripInfix [0,0] [1,0,0,2,0,0,3]
Some ([1], [2,0,0,3])

>>> stripInfix [0,0] [1,2,0,4,5]
None
```

isPrefixOf : *Eq* a => [a] -> [a] -> *Bool*
The `isPrefixOf` function takes two lists and returns `True` if and only if the first is a prefix of the second.

isSuffixOf : *Eq* a => [a] -> [a] -> *Bool*
The `isSuffixOf` function takes two lists and returns `True` if and only if the first list is a suffix of the second.

isInfixOf : *Eq* a => [a] -> [a] -> *Bool*
The `isInfixOf` function takes two lists and returns `True` if and only if the first list is contained anywhere within the second.

mapAccumL : (acc -> x -> (acc, y)) -> acc -> [x] -> (acc, [y])
The `mapAccumL` function combines the behaviours of `map` and `foldl`; it applies a function to each element of a list, passing an accumulating parameter from left to right, and returning a final value of this accumulator together with the new list.

inits : [a] -> [[a]]
The `inits` function returns all initial segments of the argument, shortest first.

intersperse : a -> [a] -> [a]
The `intersperse` function takes an element and a list and "intersperses" that element between the elements of the list.

intercalate : [a] -> [[a]] -> [a]
`intercalate` inserts the list `xs` in between the lists in `xss` and concatenates the result.

tails : [a] -> [[a]]
The `tails` function returns all final segments of the argument, longest first.

dropWhileEnd : (a -> *Bool*) -> [a] -> [a]
A version of `dropWhile` operating from the end.

takeWhileEnd : (a -> *Bool*) -> [a] -> [a]
A version of `takeWhile` operating from the end.

transpose : [[a]] -> [[a]]
The `transpose` function transposes the rows and columns of its argument.

breakEnd : (a -> *Bool*) -> [a] -> ([a], [a])
Break, but from the end.

breakOn : *Eq* a => [a] -> [a] -> ([a], [a])
Find the first instance of `needle` in `haystack`. The first element of the returned tuple is the prefix of `haystack` before `needle` is matched. The second is the remainder of `haystack`, starting with the match. If you want the remainder *without* the match, use `stripInfix`.

breakOnEnd : *Eq* a => [a] -> [a] -> ([a], [a])
Similar to `breakOn`, but searches from the end of the string.
The first element of the returned tuple is the prefix of `haystack` up to and including the last match of `needle`. The second is the remainder of `haystack`, following the match.

linesBy : (a -> *Bool*) -> [a] -> [[a]]
A variant of `lines` with a custom test. In particular, if there is a trailing separator it will be discarded.

wordsBy : (a -> *Bool*) -> [a] -> [[a]]
A variant of `words` with a custom test. In particular, adjacent separators are discarded, as are leading or trailing separators.

head : [a] -> a
Extract the first element of a list, which must be non-empty.

tail : [a] -> [a]
Extract the elements after the head of a list, which must be non-empty.

last : [a] -> a
Extract the last element of a list, which must be finite and non-empty.

init : [a] -> [a]
Return all the elements of a list except the last one. The list must be non-empty.

foldl1 : (a -> a -> a) -> [a] -> a
Left associative fold of a list that must be non-empty.

foldr1 : (a -> a -> a) -> [a] -> a
Right associative fold of a list that must be non-empty.

repeatedly : ([a] -> (b, [a])) -> [a] -> [b]
Apply some operation repeatedly, producing an element of output and the remainder of the list.

chunksOf : *Int* -> [a] -> [[a]]
Splits a list into chunks of length @n@. @n@ must be strictly greater than zero. The last chunk will be shorter than @n@ in case the length of the input is not divisible by @n@.

delete : *Eq* a => a -> [a] -> [a]
`delete x` removes the first occurrence of `x` from its list argument. For example,

```
> delete "a" ["b","a","n","a","n","a"]
["b","n","a","n","a"]
```

It is a special case of 'deleteBy', which allows the programmer to supply their own equality test.

deleteBy : (a -> a -> *Bool*) -> a -> [a] -> [a]
The 'deleteBy' function behaves like 'delete', but takes a user-supplied equality predicate.

```
> deleteBy (<=) 4 [1..10]
[1,2,3,5,6,7,8,9,10]
```

(\\\\) : *Eq* a => [a] -> [a] -> [a]
The `\\` function is list difference (non-associative). In the result of `xs \\ ys`, the first occurrence of each element of `ys` in turn (if any) has been removed from `xs`. Thus

```
(xs ++ ys) \\ xs == ys
```

Note this function is *O(n*m)* given lists of size *n* and *m*.

singleton : a -> [a]
Produce a singleton list.

```
>>> singleton True
[True]
```

(!!) : [a] -> *Int* -> a
List index (subscript) operator, starting from 0. For example, `xs !! 2` returns the third element in `xs`. Raises an error if the index is not suitable for the given list. The function has complexity *O(n)* where *n* is the index given, unlike in languages such as Java where array indexing is *O(1)*.

elemIndex : *Eq* a => a -> [a] -> *Optional Int*
Find index of element in given list. Will return `None` if not found.

findIndex : (a -> *Bool*) -> [a] -> *Optional Int*
Find index, given predicate, of first matching element. Will return `None` if not found.

1.43.3.16 DA.List.BuiltinOrder

Note: This is only supported in Daml-LF 1.11 or later.

This module provides variants of other standard library functions that are based on the builtin Daml-LF ordering rather than user-defined ordering. This is the same order also used by `DA.Map`.

These functions are usually much more efficient than their `Ord`-based counterparts.

Note that the functions in this module still require `Ord` constraints. This is purely to enforce that you don't pass in values that cannot be compared, e.g., functions. The implementation of those instances is not used.

Functions

dedup : *Ord* a => [a] -> [a]
`dedup l` removes duplicate elements from a list. In particular, it keeps only the first occurrence of each element.
`dedup` is stable so the elements in the output are ordered by their first occurrence in the input. If you do not need stability, consider using `dedupSort` which is more efficient.

```
>>> dedup [3, 1, 1, 3]
[3, 1]
```

dedupOn : *Ord* k => (v -> k) -> [v] -> [v]
A version of `dedup` where deduplication is done after applying the given function. Example use: `dedupOn (.employeeNo) employees`.
`dedupOn` is stable so the elements in the output are ordered by their first occurrence in the input. If you do not need stability, consider using `dedupOnSort` which is more efficient.

```
>>> dedupOn fst [(3, "a"), (1, "b"), (1, "c"), (3, "d")]
[(3, "a"), (1, "b")]
```

dedupSort : *Ord* a => [a] -> [a]
`dedupSort` is a more efficient variant of `dedup` that does not preserve the order of the input elements. Instead the output will be sorted acoording to the builtin Daml-LF ordering.

```
>>> dedupSort [3, 1, 1, 3]
[1, 3]
```

dedupOnSort : *Ord* k => (v -> k) -> [v] -> [v]
`dedupOnSort` is a more efficient variant of `dedupOn` that does not preserve the order of the input elements. Instead the output will be sorted on the values returned by the function.
For duplicates, the first element in the list will be included in the output.

```
>>> dedupOnSort fst [(3, "a"), (1, "b"), (1, "c"), (3, "d")]
[(1, "b"), (3, "a")]
```

sort : *Ord* a => [a] -> [a]
Sort the list according to the Daml-LF ordering.
Values that are identical according to the builtin Daml-LF ordering are indistinguishable so stability is not relevant here.

```
>>> sort [3,1,2]
[1,2,3]
```

sortOn : *Ord* b => (a -> b) -> [a] -> [a]
`sortOn f` is a version of sort that allows sorting on the result of the given function.
`sortOn` is stable so elements that map to the same sort key will be ordered by their position in the input.

```
>>> sortOn fst [(3, "a"), (1, "b"), (3, "c"), (2, "d")]
[(1, "b"), (2, "d"), (3, "a"), (3, "c")]
```

unique : *Ord* a => [a] -> *Bool*
Returns True if and only if there are no duplicate elements in the given list.

```
>>> unique [1, 2, 3]
True
```

uniqueOn : *Ord* k => (a -> k) -> [a] -> *Bool*
Returns True if and only if there are no duplicate elements in the given list after applyng function.

```
>>> uniqueOn fst [(1, 2), (2, 42), (1, 3)]
False
```

1.43.3.17 DA.List.Total

Functions

head : [a] -> *Optional* a
Return the first element of a list. Return `None` if list is empty.

tail : [a] -> *Optional* [a]
Return all but the first element of a list. Return `None` if list is empty.

last : [a] -> *Optional* a
Extract the last element of a list. Returns `None` if list is empty.

init : [a] -> *Optional* [a]
Return all the elements of a list except the last one. Returns `None` if list is empty.

(!!) : [a] -> *Int* -> *Optional* a
Return the nth element of a list. Return `None` if index is out of bounds.

foldl1 : (a -> a -> a) -> [a] -> *Optional* a
Fold left starting with the head of the list. For example, `foldl1 f [a,b,c] = f (f a b) c`. Return `None` if list is empty.

foldr1 : (a -> a -> a) -> [a] -> *Optional* a
Fold right starting with the last element of the list. For example, `foldr1 f [a,b,c] = f a (f b c)`

foldBalanced1 : (a -> a -> a) -> [a] -> *Optional* a
Fold a non-empty list in a balanced way. Balanced means that each element has approximately the same depth in the operator tree. Approximately the same depth means that the difference between maximum and minimum depth is at most 1. The accumulation operation must be associative and commutative in order to get the same result as `foldl1` or `foldr1`.
Return `None` if list is empty.

minimumBy : (a -> a -> *Ordering*) -> [a] -> *Optional* a
Return the least element of a list according to the given comparison function. Return `None` if list is empty.

maximumBy : (a -> a -> *Ordering*) -> [a] -> *Optional* a
Return the greatest element of a list according to the given comparison function. Return `None` if list is empty.

minimumOn : *Ord* k => (a -> k) -> [a] -> *Optional* a
Return the least element of a list when comparing by a key function. For example `minimumOn (\(x,y) -> x + y) [(1,2), (2,0)] == Some (2,0)`. Return `None` if list is empty.

maximumOn : *Ord* k => (a -> k) -> [a] -> *Optional* a
Return the greatest element of a list when comparing by a key function. For example `maximumOn (\(x,y) -> x + y) [(1,2), (2,0)] == Some (1,2)`. Return `None` if list is empty.

1.43.3.18 DA.Logic

Logic - Propositional calculus.

Data Types

data *Formula* t

A `Formula t` is a formula in propositional calculus with propositions of type t.

Proposition t

`Proposition p` is the formula p

Negation (*Formula* t)

For a formula f, `Negation f` is f

Conjunction [*Formula* t]

For formulas f1, ..., fn, `Conjunction [f1, ..., fn]` is f1 ... fn

Disjunction [*Formula* t]

For formulas f1, ..., fn, `Disjunction [f1, ..., fn]` is f1 ... fn

instance *Action Formula*

instance *Applicative Formula*

instance *Functor Formula*

instance *Eq* t => *Eq* (*Formula* t)

instance *Ord* t => *Ord* (*Formula* t)

instance *Show* t => *Show* (*Formula* t)

Functions

(&&&) : *Formula* t -> *Formula* t -> *Formula* t
`&&&` is the operation of the boolean algebra of formulas, to be read as "and"

(|||) : *Formula* t -> *Formula* t -> *Formula* t
`|||` is the operation of the boolean algebra of formulas, to be read as "or"

true : *Formula* t
`true` is the 1 element of the boolean algebra of formulas, represented as an empty conjunction.

false : *Formula* t
`false` is the 0 element of the boolean algebra of formulas, represented as an empty disjunction.

neg : *Formula* t -> *Formula* t
`neg` is the (negation) operation of the boolean algebra of formulas.

conj : [*Formula* t] -> *Formula* t
`conj` is a list version of `&&&`, enabled by the associativity of .

disj : [*Formula* t] -> *Formula* t
`disj` is a list version of `|||`, enabled by the associativity of .

fromBool : *Bool* -> *Formula* t
`fromBool` converts `True` to `true` and `False` to `false`.

toNNF : *Formula* t -> *Formula* t
`toNNF` transforms a formula to negation normal form (see https://en.wikipedia.org/wiki/Negation_normal_form).

toDNF : *Formula* t -> *Formula* t
`toDNF` turns a formula into disjunctive normal form. (see https://en.wikipedia.org/wiki/Disjunctive_normal_form).

traverse : *Applicative* f => (t -> f s) -> *Formula* t -> f (*Formula* s)
An implementation of `traverse` in the usual sense.

zipFormulas : *Formula* t -> *Formula* s -> *Formula* (t, s)
`zipFormulas` takes to formulas of same shape, meaning only propositions are different and zips them up.

substitute : (t -> *Optional Bool*) -> *Formula* t -> *Formula* t
`substitute` takes a truth assignment and substitutes `True` or `False` into the respective places in a formula.

reduce : *Formula* t -> *Formula* t
`reduce` reduces a formula as far as possible by:

1. Removing any occurrences of `true` and `false`;
2. Removing directly nested Conjunctions and Disjunctions;
3. Going to negation normal form.

isBool : *Formula* t -> *Optional Bool*
`isBool` attempts to convert a formula to a bool. It satisfies `isBool true == Some True` and `isBool false == Some False`. Otherwise, it returns `None`.

interpret : (t -> *Optional Bool*) -> *Formula* t -> *Either* (*Formula* t) *Bool*
`interpret` is a version of `toBool` that first substitutes using a truth function and then reduces as far as possible.

substituteA : *Applicative* f => (t -> f (*Optional Bool*)) -> *Formula* t -> f (*Formula* t)
`substituteA` is a version of `substitute` that allows for truth values to be obtained from an action.

interpretA : *Applicative* f => (t -> f (*Optional Bool*)) -> *Formula* t -> f (*Either* (*Formula* t) *Bool*)
`interpretA` is a version of `interpret` that allows for truth values to be obtained form an action.

1.43.3.19 DA.Map

Note: This is only supported in Daml-LF 1.11 or later.

This module exports the generic map type `Map k v` and associated functions. This module should be imported qualified, for example:

```
import DA.Map (Map)
import DA.Map qualified as M
```

This will give access to the `Map` type, and the various operations as `M.lookup`, `M.insert`, `M.fromList`, etc.

`Map k v` internally uses the built-in order for the type `k`. This means that keys that contain functions are not comparable and will result in runtime errors. To prevent this, the `Ord k` instance is required for most map operations. It is recommended to only use `Map k v` for key types that have an `Ord k` instance that is derived automatically using `deriving`:

```
data K = ...
  deriving (Eq, Ord)
```

This includes all built-in types that aren't function types, such as `Int`, `Text`, `Bool`, `(a, b)` assuming `a` and `b` have default `Ord` instances, `Optional t` and `[t]` assuming `t` has a default `Ord` instance, `Map k v` assuming `k` and `v` have default `Ord` instances, and `Set k` assuming `k` has a default `Ord` instance.

Functions

fromList : *Ord* k => [(k, v)] -> *Map* k v
Create a map from a list of key/value pairs.

fromListWith : *Ord* k => (v -> v -> v) -> [(k, v)] -> *Map* k v
Create a map from a list of key/value pairs with a combining function. Examples:

```
>>> fromListWith (++) [("A", [1]), ("A", [2]), ("B", [2]), ("B", [1]), ("A",
↪[3])]
fromList [("A", [1, 2, 3]), ("B", [2, 1])]
>>> fromListWith (++) [] == (empty : Map Text [Int])
True
```

keys : *Map* k v -> [k]

Get the list of keys in the map. Keys are sorted according to the built-in order for the type `k`, which matches the `Ord k` instance when using `deriving Ord`.

```
>>> keys (fromList [("A", 1), ("C", 3), ("B", 2)])
["A", "B", "C"]
```

values : *Map* k v -> [v]
Get the list of values in the map. These will be in the same order as their respective keys from `M.keys`.

```
>>> values (fromList [("A", 1), ("B", 2)])
[1, 2]
```

toList : *Map* k v -> [(k, v)]
Convert the map to a list of key/value pairs. These will be ordered by key, as in `M.keys`.

empty : *Map* k v
The empty map.

size : *Map* k v -> *Int*
Number of elements in the map.

null : *Map* k v -> *Bool*
Is the map empty?

lookup : *Ord* k => k -> *Map* k v -> *Optional* v
Lookup the value at a key in the map.

member : *Ord* k => k -> *Map* k v -> *Bool*
Is the key a member of the map?

filter : *Ord* k => (v -> *Bool*) -> *Map* k v -> *Map* k v
Filter the `Map` using a predicate: keep only the entries where the value satisfies the predicate.

filterWithKey : *Ord* k => (k -> v -> *Bool*) -> *Map* k v -> *Map* k v
Filter the `Map` using a predicate: keep only the entries which satisfy the predicate.

delete : *Ord* k => k -> *Map* k v -> *Map* k v
Delete a key and its value from the map. When the key is not a member of the map, the original map is returned.

insert : *Ord* k => k -> v -> *Map* k v -> *Map* k v
Insert a new key/value pair in the map. If the key is already present in the map, the associated value is replaced with the supplied value.

insertWith : *Ord* k => (v -> v -> v) -> k -> v -> *Map* k v -> *Map* k v
Insert a new key/value pair in the map. If the key is already present in the map, it is combined with the previous value using the given function `f new_value old_value`.

alter : *Ord* k => (*Optional* v -> *Optional* v) -> k -> *Map* k v -> *Map* k v
Update the value in `m` at `k` with `f`, inserting or deleting as required. `f` will be called with either the value at `k`, or `None` if absent; `f` can return `Some` with a new value to be inserted in `m` (replacing the old value if there was one), or `None` to remove any `k` association `m` may have.
Some implications of this behavior:
alter identity k = identity alter g k . alter f k = alter (g . f) k alter (_ -> Some v) k = insert k v alter (_ -> None) = delete

union : *Ord* k => *Map* k v -> *Map* k v -> *Map* k v

The union of two maps, preferring the first map when equal keys are encountered.

unionWith : *Ord* k => (v -> v -> v) -> *Map* k v -> *Map* k v -> *Map* k v
The union of two maps using the combining function to merge values that exist in both maps.

merge : *Ord* k => (k -> a -> *Optional* c) -> (k -> b -> *Optional* c) -> (k -> a -> b -> *Optional* c) -> *Map* k a -> *Map* k b -> *Map* k c
Combine two maps, using separate functions based on whether a key appears only in the first map, only in the second map, or appears in both maps.

1.43.3.20 DA.Math

Math - Utility Math functions for `Decimal` The this library is designed to give good precision, typically giving 9 correct decimal places. The numerical algorithms run with many iterations to achieve that precision and are interpreted by the Daml runtime so they are not performant. Their use is not advised in performance critical contexts.

Functions

(**) : *Decimal* -> *Decimal* -> *Decimal*
Take a power of a number Example: `2.0 ** 3.0 == 8.0`.

sqrt : *Decimal* -> *Decimal*
Calculate the square root of a Decimal.

```
>>> sqrt 1.44
1.2
```

exp : *Decimal* -> *Decimal*
The exponential function. Example: `exp 0.0 == 1.0`

log : *Decimal* -> *Decimal*
The natural logarithm. Example: `log 10.0 == 2.30258509299`

logBase : *Decimal* -> *Decimal* -> *Decimal*
The logarithm of a number to a given base. Example: `log 10.0 100.0 == 2.0`

sin : *Decimal* -> *Decimal*
`sin` is the sine function

cos : *Decimal* -> *Decimal*
`cos` is the cosine function

tan : *Decimal* -> *Decimal*
`tan` is the tangent function

1.43.3.21 DA.Monoid

Data Types

data *All*

Boolean monoid under conjunction (&&)

All

Field	Type	Description
getAll	*Bool*	

instance *Monoid All*

instance *Semigroup All*

instance *Eq All*

instance *Ord All*

instance *Show All*

data *Any*

Boolean Monoid under disjunction (||)

Any

Field	Type	Description
getAny	*Bool*	

instance *Monoid Any*

instance *Semigroup Any*

instance *Eq Any*

instance *Ord Any*

instance *Show Any*

data *Endo* a

The monoid of endomorphisms under composition.

Endo

Field	Type	Description
appEndo	a -> a	

instance *Monoid* (*Endo* a)

instance *Semigroup* (*Endo* a)

data *Product* a

Monoid under (*)

```
> Product 2 <> Product 3
Product 6
```

Product a

instance *Multiplicative* a => *Monoid* (*Product* a)

instance *Multiplicative* a => *Semigroup* (*Product* a)

instance *Eq* a => *Eq* (*Product* a)

instance *Ord* a => *Ord* (*Product* a)

instance *Additive* a => *Additive* (*Product* a)

instance *Multiplicative* a => *Multiplicative* (*Product* a)

instance *Show* a => *Show* (*Product* a)

data *Sum* a

Monoid under (+)

```
> Sum 1 <> Sum 2
Sum 3
```

Sum a

instance *Additive* a => *Monoid* (*Sum* a)

instance *Additive* a => *Semigroup* (*Sum* a)

instance *Eq* a => *Eq* (*Sum* a)

instance *Ord* a => *Ord* (*Sum* a)

instance *Additive* a => *Additive* (*Sum* a)

instance *Multiplicative* a => *Multiplicative* (*Sum* a)

instance *Show* a => *Show* (*Sum* a)

1.43.3.22 DA.NonEmpty

Type and functions for non-empty lists. This module re-exports many functions with the same name as prelude list functions, so it is expected to import the module qualified. For example, with the following import list you will have access to the `NonEmpty` type and any functions on non-empty lists will be qualified, for example as `NE.append`, `NE.map`, `NE.foldl`:

```
import DA.NonEmpty (NonEmpty)
import qualified DA.NonEmpty as NE
```

Functions

cons : a -> *NonEmpty* a -> *NonEmpty* a
Prepend an element to a non-empty list.

append : *NonEmpty* a -> *NonEmpty* a -> *NonEmpty* a
Append or concatenate two non-empty lists.

map : (a -> b) -> *NonEmpty* a -> *NonEmpty* b
Apply a function over each element in the non-empty list.

nonEmpty : [a] -> *Optional* (*NonEmpty* a)
Turn a list into a non-empty list, if possible. Returns `None` if the input list is empty, and `Some` otherwise.

singleton : a -> *NonEmpty* a
A non-empty list with a single element.

toList : *NonEmpty* a -> [a]
Turn a non-empty list into a list (by forgetting that it is not empty).

reverse : *NonEmpty* a -> *NonEmpty* a
Reverse a non-empty list.

find : (a -> *Bool*) -> *NonEmpty* a -> *Optional* a
Find an element in a non-empty list.

deleteBy : (a -> a -> *Bool*) -> a -> *NonEmpty* a -> [a]
The 'deleteBy' function behaves like 'delete', but takes a user-supplied equality predicate.

delete : *Eq* a => a -> *NonEmpty* a -> [a]
Remove the first occurence of x from the non-empty list, potentially removing all elements.

foldl1 : (a -> a -> a) -> *NonEmpty* a -> a
Apply a function repeatedly to pairs of elements from a non-empty list, from the left. For example, `foldl1 (+) (NonEmpty 1 [2,3,4]) = ((1 + 2) + 3) + 4`.

foldr1 : (a -> a -> a) -> *NonEmpty* a -> a
Apply a function repeatedly to pairs of elements from a non-empty list, from the right. For example, `foldr1 (+) (NonEmpty 1 [2,3,4]) = 1 + (2 + (3 + 4))`.

foldr : (a -> b -> b) -> b -> *NonEmpty* a -> b
Apply a function repeatedly to pairs of elements from a non-empty list, from the right, with a given initial value. For example, `foldr (+) 0 (NonEmpty 1 [2,3,4]) = 1 + (2 + (3 + (4 + 0)))`.

foldrA : *Action* m => (a -> b -> m b) -> b -> *NonEmpty* a -> m b
The same as `foldr` but running an action each time.

foldr1A : *Action* m => (a -> a -> m a) -> *NonEmpty* a -> m a
The same as `foldr1` but running an action each time.

foldl : (b -> a -> b) -> b -> *NonEmpty* a -> b
Apply a function repeatedly to pairs of elements from a non-empty list, from the left, with a given initial value. For example, `foldl (+) 0 (NonEmpty 1 [2,3,4]) = (((0 + 1) + 2) + 3) + 4`.

foldlA : *Action* m => (b -> a -> m b) -> b -> *NonEmpty* a -> m b
The same as `foldl` but running an action each time.

foldl1A : *Action* m => (a -> a -> m a) -> *NonEmpty* a -> m a
The same as `foldl1` but running an action each time.

1.43.3.23 DA.NonEmpty.Types

This module contains the type for non-empty lists so we can give it a stable package id. This is reexported from DA.NonEmpty so you should never need to import this module.

Data Types

data *NonEmpty* a

`NonEmpty` is the type of non-empty lists. In other words, it is the type of lists that always contain at least one element. If `x` is a non-empty list, you can obtain the first element with `x.hd` and the rest of the list with `x.tl`.

NonEmpty

Field	Type	Description
hd	a	
tl	[a]	

instance *Foldable NonEmpty*

instance *Action NonEmpty*

instance *Applicative NonEmpty*

instance *Semigroup* (*NonEmpty* a)

instance *IsParties* (*NonEmpty Party*)

instance *Traversable NonEmpty*

instance *Functor NonEmpty*

instance *Eq* a => *Eq* (*NonEmpty* a)

instance *Ord* a => *Ord* (*NonEmpty* a)

instance *Show* a => *Show* (*NonEmpty* a)

1.43.3.24 DA.Numeric

Functions

mul : *NumericScale* n3 => *Numeric* n1 -> *Numeric* n2 -> *Numeric* n3
Multiply two numerics. Both inputs and the output may have different scales, unlike `(*)` which forces all numeric scales to be the same. Raises an error on overflow, rounds to chosen scale otherwise.

div : *NumericScale* n3 => *Numeric* n1 -> *Numeric* n2 -> *Numeric* n3
Divide two numerics. Both inputs and the output may have different scales, unlike `(/)` which forces all numeric scales to be the same. Raises an error on overflow, rounds to chosen scale otherwise.

cast : *NumericScale* n2 => *Numeric* n1 -> *Numeric* n2
Cast a Numeric. Raises an error on overflow or loss of precision.

castAndRound : *NumericScale* n2 => *Numeric* n1 -> *Numeric* n2
Cast a Numeric. Raises an error on overflow, rounds to chosen scale otherwise.

shift : *NumericScale* n2 => *Numeric* n1 -> *Numeric* n2
Move the decimal point left or right by multiplying the numeric value by 10^(n1 - n2). Does not overflow or underflow.

pi : *NumericScale* n => *Numeric* n
The number pi.

1.43.3.25 DA.Optional

The `Optional` type encapsulates an optional value. A value of type `Optional a` either contains a value of type `a` (represented as `Some a`), or it is empty (represented as `None`). Using `Optional` is a good way to deal with errors or exceptional cases without resorting to drastic measures such as error.

The Optional type is also an action. It is a simple kind of error action, where all errors are represented by `None`. A richer error action can be built using the `Either` type.

Functions

fromSome : *Optional* a -> a
The `fromSome` function extracts the element out of a `Some` and throws an error if its argument is `None`.
Note that in most cases you should prefer using `fromSomeNote` to get a better error on failures.

fromSomeNote : *Text* -> *Optional* a -> a
Like `fromSome` but with a custom error message.

catOptionals : [*Optional* a] -> [a]
The `catOptionals` function takes a list of `Optionals` and returns a list of all the `Some` values.

listToOptional : [a] -> *Optional* a
The `listToOptional` function returns `None` on an empty list or `Some` a where a is the first element of the list.

optionalToList : *Optional* a -> [a]
The `optionalToList` function returns an empty list when given `None` or a singleton list when not given `None`.

fromOptional : a -> *Optional* a -> a
The `fromOptional` function takes a default value and a `Optional` value. If the `Optional` is `None`, it returns the default values otherwise, it returns the value contained in the `Optional`.

isSome : *Optional* a -> *Bool*
The `isSome` function returns `True` iff its argument is of the form `Some _`.

isNone : *Optional* a -> *Bool*
The `isNone` function returns `True` iff its argument is `None`.

mapOptional : (a -> *Optional* b) -> [a] -> [b]
The `mapOptional` function is a version of `map` which can throw out elements. In particular, the functional argument returns something of type `Optional b`. If this is `None`, no element is added on to the result list. If it is `Some b`, then `b` is included in the result list.

whenSome : *Applicative* m => *Optional* a -> (a -> m ()) -> m ()
Perform some operation on `Some`, given the field inside the `Some`.

findOptional : (a -> *Optional* b) -> [a] -> *Optional* b
The `findOptional` returns the value of the predicate at the first element where it returns `Some`. `findOptional` is similar to `find` but it allows you to return a value from the predicate. This is useful both as a more type safe version if the predicate corresponds to a pattern match and for performance to avoid duplicating work performed in the predicate.

1.43.3.26 DA.Record

Exports the record machinery necessary to allow one to annotate code that is polymorphic in the underlying record type.

Typeclasses

class *HasField* x r a **where**

`HasField` gives you getter and setter functions for each record field automatically.

In the vast majority of use-cases, plain Record syntax should be preferred:

```
daml> let a = MyRecord 1 "hello"
daml> a.foo
1
daml> a.bar
"hello"
daml> a { bar = "bye" }
MyRecord {foo = 1, bar = "bye"}
daml> a with foo = 3
MyRecord {foo = 3, bar = "hello"}
daml>
```

For more on Record syntax, see https://docs.daml.com/daml/intro/3_Data.html#record.

`HasField x r a` is a typeclass that takes three parameters. The first parameter `x` is the field name, the second parameter `r` is the record type, and the last parameter `a` is the type of the field in this record. For example, if we define a type:

```
data MyRecord = MyRecord with
    foo : Int
    bar : Text
```

Then we get, for free, the following HasField instances:

```
HasField "foo" MyRecord Int
HasField "bar" MyRecord Text
```

If we want to get a value using HasField, we can use the `getField` function:

```
getFoo : MyRecord -> Int
getFoo r = getField @"foo" r

getBar : MyRecord -> Text
getBar r = getField @"bar" r
```

Note that this uses the type application syntax (`f @t`) to specify the field name.

Likewise, if we want to set the value in the field, we can use the `setField` function:

```
setFoo : Int -> MyRecord -> MyRecord
setFoo a r = setField @"foo" a r

setBar : Text -> MyRecord -> MyRecord
setBar a r = setField @"bar" a r
```

getField : r -> a

setField : a -> r -> r

1.43.3.27 DA.Semigroup

Data Types

data *Max* a

Semigroup under `max`

```
> Max 23 <> Max 42
Max 42
```

Max a

instance *Ord* a => *Semigroup* (*Max* a)

instance *Eq* a => *Eq* (*Max* a)

instance *Ord* a => *Ord* (*Max* a)

instance *Show* a => *Show* (*Max* a)

data *Min* a

Semigroup under `min`

```
> Min 23 <> Min 42
Min 23
```

Min a

instance *Ord* a => *Semigroup* (*Min* a)

instance *Eq* a => *Eq* (*Min* a)

instance *Ord* a => *Ord* (*Min* a)

instance *Show* a => *Show* (*Min* a)

1.43.3.28 DA.Set

Note: This is only supported in Daml-LF 1.11 or later.

This module exports the generic set type `Set k` and associated functions. This module should be imported qualified, for example:

```
import DA.Set (Set)
import DA.Set qualified as S
```

This will give access to the `Set` type, and the various operations as `S.lookup`, `S.insert`, `S.fromList`, etc.

`Set k` internally uses the built-in order for the type `k`. This means that keys that contain functions are not comparable and will result in runtime errors. To prevent this, the `Ord k` instance is required for most set operations. It is recommended to only use `Set k` for key types that have an `Ord k` instance that is derived automatically using `deriving`:

```
data K = ...
  deriving (Eq, Ord)
```

This includes all built-in types that aren't function types, such as `Int`, `Text`, `Bool`, `(a, b)` assuming `a` and `b` have default `Ord` instances, `Optional t` and `[t]` assuming `t` has a default `Ord` instance, `Map k v` assuming `k` and `v` have default `Ord` instances, and `Set k` assuming `k` has a default `Ord` instance.

Data Types

data *Set* k

The type of a set. This is a wrapper over the `Map` type.

Set

Field	Type	Description
map	*Map* k ()	

instance *Foldable Set*

instance *Ord* k => *Monoid* (*Set* k)

instance *Ord* k => *Semigroup* (*Set* k)

instance *IsParties* (*Set Party*)

instance *Ord* k => *Eq* (*Set* k)

instance *Ord* k => *Ord* (*Set* k)

instance (*Ord* k, *Show* k) => *Show* (*Set* k)

Functions

empty : *Set* k
The empty set.

size : *Set* k -> *Int*
The number of elements in the set.

toList : *Set* k -> [k]
Convert the set to a list of elements.

fromList : *Ord* k => [k] -> *Set* k
Create a set from a list of elements.

toMap : *Set* k -> *Map* k ()
Convert a `Set` into a `Map`.

fromMap : *Map* k () -> *Set* k
Create a `Set` from a `Map`.

member : *Ord* k => k -> *Set* k -> *Bool*
Is the element in the set?

notMember : *Ord* k => k -> *Set* k -> *Bool*
Is the element not in the set? `notMember k s` is equivalent to `not (member k s)`.

null : *Set* k -> *Bool*
Is this the empty set?

insert : *Ord* k => k -> *Set* k -> *Set* k
Insert an element in a set. If the set already contains the element, this returns the set unchanged.

filter : *Ord* k => (k -> *Bool*) -> *Set* k -> *Set* k
Filter all elements that satisfy the predicate.

delete : *Ord* k => k -> *Set* k -> *Set* k
Delete an element from a set.

singleton : *Ord* k => k -> *Set* k
Create a singleton set.

union : *Ord* k => *Set* k -> *Set* k -> *Set* k
The union of two sets.

intersection : *Ord* k => *Set* k -> *Set* k -> *Set* k
The intersection of two sets.

difference : *Ord* k => *Set* k -> *Set* k -> *Set* k
`difference x y` returns the set consisting of all elements in `x` that are not in `y`.
> > > fromList [1, 2, 3] *difference* fromList [1, 4] > > > fromList [2, 3]

isSubsetOf : *Ord* k => *Set* k -> *Set* k -> *Bool*
`isSubsetOf a b` returns true if `a` is a subset of `b`, that is, if every element of `a` is in `b`.

isProperSubsetOf : *Ord* k => *Set* k -> *Set* k -> *Bool*
`isProperSubsetOf a b` returns true if `a` is a proper subset of `b`. That is, if `a` is a subset of `b` but not equal to `b`.

1.43.3.29 DA.Stack

Data Types

data *SrcLoc*

Location in the source code.

Line and column are 0-based.

SrcLoc

Field	Type	Description
srcLocPackage	*Text*	
srcLocModule	*Text*	
srcLocFile	*Text*	
srcLocStartLine	*Int*	
srcLocStartCol	*Int*	
srcLocEndLine	*Int*	
srcLocEndCol	*Int*	

data *CallStack*

Type of callstacks constructed automatically from `HasCallStack` constraints.

Use `callStack` to get the current callstack, and use `getCallStack` to deconstruct the `CallStack`.

type ***HasCallStack*** = IP "callStack" *CallStack*

Request a `CallStack`. Use this as a constraint in type signatures in order to get nicer callstacks for error and debug messages.
For example, instead of declaring the following type signature:

```
myFunction : Int -> Update ()
```

You can declare a type signature with the `HasCallStack` constraint:

```
myFunction : HasCallStack => Int -> Update ()
```

The function `myFunction` will still be called the same way, but it will also show up as an entry in the current callstack, which you can obtain with `callStack`.
Note that only functions with the `HasCallStack` constraint will be added to the current callstack, and if any function does not have the `HasCallStack` constraint, the callstack will be reset within that function.

Functions

prettyCallStack : *CallStack* -> *Text*
Pretty-print a `CallStack`.

getCallStack : *CallStack* -> [(*Text*, *SrcLoc*)]
Extract the list of call sites from the `CallStack`.
The most recent call comes first.

callStack : *HasCallStack* => *CallStack*
Access to the current `CallStack`.

1.43.3.30 DA.Text

Functions for working with Text.

Functions

explode : *Text* -> [*Text*]

implode : [*Text*] -> *Text*

isEmpty : *Text* -> *Bool*
Test for emptiness.

length : *Text* -> *Int*
Compute the number of symbols in the text.

trim : *Text* -> *Text*
Remove spaces from either side of the given text.

replace : *Text* -> *Text* -> *Text* -> *Text*
Replace a subsequence everywhere it occurs. The first argument must not be empty.

lines : *Text* -> [*Text*]
Breaks a `Text` value up into a list of `Text`'s at newline symbols. The resulting texts do not contain newline symbols.

unlines : [*Text*] -> *Text*
Joins lines, after appending a terminating newline to each.

words : *Text* -> [*Text*]
Breaks a 'Text' up into a list of words, delimited by symbols representing white space.

unwords : [*Text*] -> *Text*
Joins words using single space symbols.

linesBy : (*Text* -> *Bool*) -> *Text* -> [*Text*]
A variant of `lines` with a custom test. In particular, if there is a trailing separator it will be discarded.

wordsBy : (*Text* -> *Bool*) -> *Text* -> [*Text*]
A variant of `words` with a custom test. In particular, adjacent separators are discarded, as are leading or trailing separators.

intercalate : *Text* -> [*Text*] -> *Text*

`intercalate` inserts the text argument `t` in between the items in `ts` and concatenates the result.

dropPrefix : *Text* -> *Text* -> *Text*

`dropPrefix` drops the given prefix from the argument. It returns the original text if the text doesn't start with the given prefix.

dropSuffix : *Text* -> *Text* -> *Text*

Drops the given suffix from the argument. It returns the original text if the text doesn't end with the given suffix. Examples:

```
dropSuffix "!" "Hello World!"  == "Hello World"
dropSuffix "!" "Hello World!!" == "Hello World!"
dropSuffix "!" "Hello World."  == "Hello World."
```

stripSuffix : *Text* -> *Text* -> *Optional Text*

Return the prefix of the second text if its suffix matches the entire first text. Examples:

```
stripSuffix "bar" "foobar" == Some "foo"
stripSuffix ""    "baz"    == Some "baz"
stripSuffix "foo" "quux"   == None
```

stripPrefix : *Text* -> *Text* -> *Optional Text*

The `stripPrefix` function drops the given prefix from the argument text. It returns `None` if the text did not start with the prefix.

isPrefixOf : *Text* -> *Text* -> *Bool*

The `isPrefixOf` function takes two text arguments and returns `True` if and only if the first is a prefix of the second.

isSuffixOf : *Text* -> *Text* -> *Bool*

The `isSuffixOf` function takes two text arguments and returns `True` if and only if the first is a suffix of the second.

isInfixOf : *Text* -> *Text* -> *Bool*

The `isInfixOf` function takes two text arguments and returns `True` if and only if the first is contained, wholly and intact, anywhere within the second.

takeWhile : (*Text* -> *Bool*) -> *Text* -> *Text*

The function `takeWhile`, applied to a predicate `p` and a text, returns the longest prefix (possibly empty) of symbols that satisfy `p`.

takeWhileEnd : (*Text* -> *Bool*) -> *Text* -> *Text*

The function 'takeWhileEnd', applied to a predicate `p` and a 'Text', returns the longest suffix (possibly empty) of elements that satisfy `p`.

dropWhile : (*Text* -> *Bool*) -> *Text* -> *Text*

`dropWhile p t` returns the suffix remaining after `takeWhile p t`.

dropWhileEnd : (*Text* -> *Bool*) -> *Text* -> *Text*

`dropWhileEnd p t` returns the prefix remaining after dropping symbols that satisfy the predicate `p` from the end of `t`.

splitOn : *Text* -> *Text* -> [*Text*]

Break a text into pieces separated by the first text argument (which cannot be empty), consuming the delimiter.

splitAt : *Int* -> *Text* -> (*Text*, *Text*)
Split a text before a given position so that for `0 <= n <= length t`, `length (fst (splitAt n t)) == n`.

take : *Int* -> *Text* -> *Text*
`take n`, applied to a text `t`, returns the prefix of `t` of length `n`, or `t` itself if `n` is greater than the length of `t`.

drop : *Int* -> *Text* -> *Text*
`drop n`, applied to a text `t`, returns the suffix of `t` after the first `n` characters, or the empty `Text` if `n` is greater than the length of `t`.

substring : *Int* -> *Int* -> *Text* -> *Text*
Compute the sequence of symbols of length `l` in the argument text starting at `s`.

isPred : (*Text* -> *Bool*) -> *Text* -> *Bool*
`isPred f t` returns `True` if `t` is not empty and `f` is `True` for all symbols in `t`.

isSpace : *Text* -> *Bool*
`isSpace t` is `True` if `t` is not empty and consists only of spaces.

isNewLine : *Text* -> *Bool*
`isSpace t` is `True` if `t` is not empty and consists only of newlines.

isUpper : *Text* -> *Bool*
`isUpper t` is `True` if `t` is not empty and consists only of uppercase symbols.

isLower : *Text* -> *Bool*
`isLower t` is `True` if `t` is not empty and consists only of lowercase symbols.

isDigit : *Text* -> *Bool*
`isDigit t` is `True` if `t` is not empty and consists only of digit symbols.

isAlpha : *Text* -> *Bool*
`isAlpha t` is `True` if `t` is not empty and consists only of alphabet symbols.

isAlphaNum : *Text* -> *Bool*
`isAlphaNum t` is `True` if `t` is not empty and consists only of alphanumeric symbols.

parseInt : *Text* -> *Optional Int*
Attempt to parse an `Int` value from a given `Text`.

parseNumeric : *Text* -> *Optional* (*Numeric* n)
Attempt to parse a `Numeric` value from a given `Text`. To get `Some` value, the text must follow the regex `(-|\+)?[0-9]+(\.[0-9]+)?` In particular, the shorthands `".12"` and `"12."` do not work, but the value can be prefixed with `+`. Leading and trailing zeros are fine, however spaces are not. Examples:

```
parseNumeric "3.14" == Some 3.14
parseNumeric "+12.0" == Some 12
```

parseDecimal : *Text* -> *Optional Decimal*
Attempt to parse a `Decimal` value from a given `Text`. To get `Some` value, the text must follow the regex `(-|\+)?[0-9]+(\.[0-9]+)?` In particular, the shorthands `".12"` and `"12."` do not work, but the value can be prefixed with `+`. Leading and trailing zeros are fine, however spaces are not. Examples:

```
parseDecimal "3.14" == Some 3.14
parseDecimal "+12.0" == Some 12
```

sha256 : *Text* -> *Text*
Computes the SHA256 hash of the UTF8 bytes of the `Text`, and returns it in its hex-encoded form. The hex encoding uses lowercase letters.
This function will crash at runtime if you compile Daml to Daml-LF < 1.2.

reverse : *Text* -> *Text*
Reverse some `Text`.

```
reverse "Daml" == "lmaD"
```

toCodePoints : *Text* -> [*Int*]
Convert a `Text` into a sequence of unicode code points.

fromCodePoints : [*Int*] -> *Text*
Convert a sequence of unicode code points into a `Text`. Raises an exception if any of the code points is invalid.

asciiToLower : *Text* -> *Text*
Convert the uppercase ASCII characters of a `Text` to lowercase; all other characters remain unchanged.

asciiToUpper : *Text* -> *Text*
Convert the lowercase ASCII characters of a `Text` to uppercase; all other characters remain unchanged.

1.43.3.31 DA.TextMap

TextMap - A map is an associative array data type composed of a collection of key/value pairs such that, each possible key appears at most once in the collection.

Functions

fromList : [(*Text*, a)] -> *TextMap* a
Create a map from a list of key/value pairs.

fromListWith : (a -> a -> a) -> [(*Text*, a)] -> *TextMap* a
Create a map from a list of key/value pairs with a combining function. Examples:

```
fromListWith (++) [("A", [1]), ("A", [2]), ("B", [2]), ("B", [1]), ("A",
↪[3])] == fromList [("A", [1, 2, 3]), ("B", [2, 1])]
fromListWith (++) [] == (empty : TextMap [Int])
```

toList : *TextMap* a -> [(*Text*, a)]
Convert the map to a list of key/value pairs where the keys are in ascending order.

empty : *TextMap* a
The empty map.

size : *TextMap* a -> *Int*
Number of elements in the map.

null : *TextMap* v -> *Bool*
Is the map empty?

lookup : *Text* -> *TextMap* a -> *Optional* a
Lookup the value at a key in the map.

member : *Text* -> *TextMap* v -> *Bool*
Is the key a member of the map?

filter : (v -> *Bool*) -> *TextMap* v -> *TextMap* v
Filter the `TextMap` using a predicate: keep only the entries where the value satisfies the predicate.

filterWithKey : (*Text* -> v -> *Bool*) -> *TextMap* v -> *TextMap* v
Filter the `TextMap` using a predicate: keep only the entries which satisfy the predicate.

delete : *Text* -> *TextMap* a -> *TextMap* a
Delete a key and its value from the map. When the key is not a member of the map, the original map is returned.

insert : *Text* -> a -> *TextMap* a -> *TextMap* a
Insert a new key/value pair in the map. If the key is already present in the map, the associated value is replaced with the supplied value.

union : *TextMap* a -> *TextMap* a -> *TextMap* a
The union of two maps, preferring the first map when equal keys are encountered.

merge : (*Text* -> a -> *Optional* c) -> (*Text* -> b -> *Optional* c) -> (*Text* -> a -> b -> *Optional* c) -> *TextMap* a -> *TextMap* b -> *TextMap* c
Merge two maps. `merge f g h x y` applies `f` to all key/value pairs whose key only appears in `x`, `g` to all pairs whose key only appears in `y` and `h` to all pairs whose key appears in both `x` and `y`. In the end, all pairs yielding `Some` are collected as the result.

1.43.3.32 DA.Time

Data Types

data *RelTime*

The `RelTime` type describes a time offset, i.e. relative time.

instance *Eq RelTime*

instance *Ord RelTime*

instance *Additive RelTime*

instance *Signed RelTime*

instance *Show RelTime*

Functions

time : *Date* -> *Int* -> *Int* -> *Int* -> *Time*
`time d h m s` turns given UTC date `d` and the UTC time (given in hours, minutes, seconds) into a UTC timestamp (`Time`). Does not handle leap seconds.

pass : *RelTime* -> *Scenario Time*
Pass simulated scenario time by argument

addRelTime : *Time* -> *RelTime* -> *Time*
Adjusts `Time` with given time offset.

subTime : *Time* -> *Time* -> *RelTime*
Returns time offset between two given instants.

wholeDays : *RelTime* -> *Int*
Returns the number of whole days in a time offset. Fraction of time is rounded towards zero.

days : *Int* -> *RelTime*
A number of days in relative time.

hours : *Int* -> *RelTime*
A number of hours in relative time.

minutes : *Int* -> *RelTime*
A number of minutes in relative time.

seconds : *Int* -> *RelTime*
A number of seconds in relative time.

milliseconds : *Int* -> *RelTime*
A number of milliseconds in relative time.

microseconds : *Int* -> *RelTime*
A number of microseconds in relative time.

convertRelTimeToMicroseconds : *RelTime* -> *Int*
Convert RelTime to microseconds Use higher level functions instead of the internal microseconds

convertMicrosecondsToRelTime : *Int* -> *RelTime*
Convert microseconds to RelTime Use higher level functions instead of the internal microseconds

1.43.3.33 DA.Traversable

Class of data structures that can be traversed from left to right, performing an action on each element. You typically would want to import this module qualified to avoid clashes with functions defined in `Prelude`. Ie.:

```
import DA.Traversable   qualified as F
```

Typeclasses

class (*Functor* t, *Foldable* t) => *Traversable* t **where**

Functors representing data structures that can be traversed from left to right.

mapA : *Applicative* f => (a -> f b) -> t a -> f (t b)
Map each element of a structure to an action, evaluate these actions from left to right, and collect the results.

sequence : *Applicative* f => t (f a) -> f (t a)
Evaluate each action in the structure from left to right, and collect the results.

instance *Ord* k => *Traversable* (*Map* k)

instance *Traversable TextMap*

instance *Traversable Optional*

instance *Traversable NonEmpty*

instance *Traversable* (*Validation* err)

instance *Traversable* (*Either* a)

instance *Traversable ([])*

instance *Traversable* a

Functions

forA : (*Traversable* t, *Applicative* f) => t a -> (a -> f b) -> f (t b)
`forA` is `mapA` with its arguments flipped.

1.43.3.34 DA.Tuple

Tuple - Ubiquitous functions of tuples.

Functions

first : (a -> a') -> (a, b) -> (a', b)
The pair obtained from a pair by application of a programmer supplied function to the argument pair's first field.

second : (b -> b') -> (a, b) -> (a, b')
The pair obtained from a pair by application of a programmer supplied function to the argument pair's second field.

both : (a -> b) -> (a, a) -> (b, b)
The pair obtained from a pair by application of a programmer supplied function to both the argument pair's first and second fields.

swap : (a, b) -> (b, a)
The pair obtained from a pair by permuting the order of the argument pair's first and second fields.

dupe : a -> (a, a)
Duplicate a single value into a pair.
> dupe 12 == (12, 12)

fst3 : (a, b, c) -> a
Extract the 'fst' of a triple.

snd3 : (a, b, c) -> b
Extract the 'snd' of a triple.

thd3 : (a, b, c) -> c
Extract the final element of a triple.

curry3 : ((a, b, c) -> d) -> a -> b -> c -> d
Converts an uncurried function to a curried function.

uncurry3 : (a -> b -> c -> d) -> (a, b, c) -> d
Converts a curried function to a function on a triple.

1.43.3.35 DA.Validation

`Validation` type and associated functions.

Data Types

data *Validation* err a

A `Validation` represents eithor a non-empty list of errors, or a successful value. This generalizes `Either` to allow more than one error to be collected.

Errors (*NonEmpty* err)

Success a

instance *Foldable* (*Validation* err)

instance *Applicative* (*Validation* err)

instance *Semigroup* (*Validation* err a)

instance *Traversable* (*Validation* err)

instance *Functor* (*Validation* err)

instance (*Eq* err, *Eq* a) => *Eq* (*Validation* err a)

instance (*Show* err, *Show* a) => *Show* (*Validation* err a)

Functions

invalid : err -> *Validation* err a
Fail for the given reason.

ok : a -> *Validation* err a
Succeed with the given value.

validate : *Either* err a -> *Validation* err a
Turn an `Either` into a `Validation`.

run : *Validation* err a -> *Either* (*NonEmpty* err) a
Convert a `Validation err a` value into an `Either`, taking the non-empty list of errors as the left value.

run1 : *Validation* err a -> *Either* err a
Convert a `Validation err a` value into an `Either`, taking just the first error as the left value.

runWithDefault : a -> *Validation* err a -> a
Run a `Validation err a` with a default value in case of errors.

(<?>) : *Optional* b -> err -> *Validation* err b
Convert an `Optional t` into a `Validation err t`, or more generally into an `m t` for any `ActionFail` type `m`.

1.43.3.36 GHC.Show.Text

Functions

showsPrecText : *Int* -> *Text* -> ShowS

1.43.3.37 GHC.Tuple.Check

Functions

userWrittenTuple : a -> a

1.43.4 Daml Script Library

The Daml Script library defines the API used to implement Daml scripts. See *Daml Script*:: for more information on Daml script.

1.43.4.1 Daml.Script

Data Types

data *Commands* a

This is used to build up the commands send as part of `submit`. If you enable the `ApplicativeDo` extension by adding `{-# LANGUAGE ApplicativeDo #-}` at the top of your file, you can use `do`-notation but the individual commands must not depend on each other and the last statement in a `do` block must be of the form `return expr` or `pure expr`.

instance Functor *Commands*

instance HasSubmit *Script Commands*

instance Applicative *Commands*

instance HasField "commands" (SubmitCmd a) (*Commands* a)

instance HasField "commands" (SubmitMustFailCmd a) (*Commands* a)

instance HasField "commands" (SubmitTreePayload a) (*Commands* ())

data *InvalidUserId*

Thrown if text for a user identifier does not conform to the format restriction.

InvalidUserId

Field	Type	Description
m	Text	

instance Eq *InvalidUserId*

instance Show *InvalidUserId*

instance HasFromAnyException *InvalidUserId*

instance HasMessage *InvalidUserId*

instance HasThrow *InvalidUserId*

instance HasToAnyException *InvalidUserId*

instance HasField "m" *InvalidUserId* Text

data *ParticipantName*

ParticipantName

Field	Type	Description
participantName	Text	

instance HasField "participantName" *ParticipantName* Text

data *PartyDetails*

The party details returned by the party management service.

PartyDetails

Field	Type	Description
party	Party	Party id
displayName	Optional Text	Optional display name
isLocal	Bool	True if party is hosted by the backing participant.

instance Eq *PartyDetails*

instance Ord *PartyDetails*

instance Show *PartyDetails*

instance HasField "continue" (ListKnownPartiesPayload a) ([*PartyDetails*] -> a)

instance HasField "displayName" *PartyDetails* (Optional Text)

instance HasField "isLocal" *PartyDetails* Bool

instance HasField "party" *PartyDetails* Party

data *PartyIdHint*

A hint to the backing participant what party id to allocate. Must be a valid PartyIdString (as described in @value.proto@).

PartyIdHint

Field	Type	Description
partyIdHint	Text	

instance HasField "partyIdHint" *PartyIdHint* Text

data *Script* a

This is the type of A Daml script. `Script` is an instance of `Action`, so you can use `do` notation.

instance Functor *Script*

instance CanAssert *Script*

instance ActionCatch *Script*

instance ActionThrow *Script*

instance CanAbort *Script*

instance HasSubmit *Script Commands*

instance HasTime *Script*

instance Action *Script*

instance ActionFail *Script*

instance Applicative *Script*

instance HasField "dummy" (*Script* a) ()

instance HasField "runScript" (*Script* a) (() -> Free ScriptF (a, ()))

data *User*

User-info record for a user in the user management service.

User

Field	Type	Description
userId	*UserId*	
primaryParty	Optional Party	

instance Eq *User*

instance Ord *User*

instance Show *User*

instance HasField "continue" (GetUserPayload a) (Optional *User* -> a)

instance HasField "continue" (ListAllUsersPayload a) ([*User*] -> a)

instance HasField "primaryParty" *User* (Optional Party)

instance HasField "user" (CreateUserPayload a) *User*

instance HasField "userId" *User UserId*

data *UserAlreadyExists*

Thrown if a user to be created already exists.

UserAlreadyExists

Field	Type	Description
userId	*UserId*	

instance Eq *UserAlreadyExists*

instance Show *UserAlreadyExists*

instance HasFromAnyException *UserAlreadyExists*

instance HasMessage *UserAlreadyExists*

instance HasThrow *UserAlreadyExists*

instance HasToAnyException *UserAlreadyExists*

instance HasField "userId" *UserAlreadyExists UserId*

data *UserId*

Identifier for a user in the user management service.

instance Eq *UserId*

instance Ord *UserId*

instance Show *UserId*

instance HasField "userId" (DeleteUserPayload a) *UserId*

instance HasField "userId" (GetUserPayload a) *UserId*

instance HasField "userId" (GrantUserRightsPayload a) *UserId*

instance HasField "userId" (ListUserRightsPayload a) *UserId*

instance HasField "userId" (RevokeUserRightsPayload a) *UserId*

instance HasField "userId" *User UserId*

instance HasField "userId" *UserAlreadyExists UserId*

instance HasField "userId" *UserNotFound UserId*

data *UserNotFound*

Thrown if a user cannot be located for a given user identifier.

UserNotFound

Field	Type	Description
userId	*UserId*	

instance Eq *UserNotFound*

instance Show *UserNotFound*

instance HasFromAnyException *UserNotFound*

instance HasMessage *UserNotFound*

instance HasThrow *UserNotFound*

instance HasToAnyException *UserNotFound*

instance HasField "userId" *UserNotFound UserId*

data *UserRight*

The rights of a user.

ParticipantAdmin

CanActAs Party

CanReadAs Party

instance Eq *UserRight*

instance Show *UserRight*

instance HasField "continue" (GrantUserRightsPayload a) (Optional [*UserRight*] -> a)

instance HasField "continue" (ListUserRightsPayload a) (Optional [*UserRight*] -> a)

instance HasField "continue" (RevokeUserRightsPayload a) (Optional [*UserRight*] -> a)

instance HasField "rights" (CreateUserPayload a) [*UserRight*]

instance HasField "rights" (GrantUserRightsPayload a) [*UserRight*]

instance HasField "rights" (RevokeUserRightsPayload a) [*UserRight*]

Functions

query : (Template t, HasAgreement t, IsParties p) => p -> *Script* [(ContractId t, t)]
Query the set of active contracts of the template that are visible to the given party.

queryFilter : (Template c, HasAgreement c, IsParties p) => p -> (c -> Bool) -> *Script* [(ContractId c, c)]
Query the set of active contracts of the template that are visible to the given party and match the given predicate.

queryContractId : (Template t, HasAgreement t, IsParties p, HasCallStack) => p -> ContractId t -> *Script* (Optional t)
Query for the contract with the given contract id.
Returns `None` if there is no active contract the party is a stakeholder on.
WARNING: Over the gRPC and with the JSON API in-memory backend this performs a linear search so only use this if the number of active contracts is small.
This is semantically equivalent to calling `query` and filtering on the client side.

queryInterface : (Template i, HasInterfaceView i v, IsParties p) => p -> *Script* [(ContractId i, Optional v)]
Query the set of active contract views for an interface that are visible to the given party. If the view function fails for a given contract id, The `Optional v` will be `None`.
WARNING: Information about instances with failed-views is not currently returned over the JSON API: the `Optional v` will be `Some _` for every element in the returned list.

queryInterfaceContractId : (Template i, HasInterfaceView i v, IsParties p, HasCallStack) => p -> ContractId i -> *Script* (Optional v)
Query for the contract view with the given contract id.
Returns `None` if there is no active contract the party is a stakeholder on.
Returns `None` if the view function fails for the given contract id.
WARNING: Over the gRPC and with the JSON API in-memory backend this performs a linear search so only use this if the number of active contracts is small.
This is semantically equivalent to calling `queryInterface` and filtering on the client side.

queryContractKey : (HasCallStack, TemplateKey t k, IsParties p) => p -> k -> *Script* (Optional (ContractId t, t))
Returns `None` if there is no active contract with the given key that the party is a stakeholder on.
WARNING: Over the gRPC and with the JSON API in-memory backend this performs a linear search so only use this if the number of active contracts is small.
This is semantically equivalent to calling `query` and filtering on the client side.

setTime : HasCallStack => Time -> *Script* ()
Set the time via the time service.
This is only supported in Daml Studio and `daml test` as well as when running over the gRPC API against a ledger in static time mode.
Note that the ledger time service does not support going backwards in time. However, you can go back in time in Daml Studio.

passTime : RelTime -> *Script* ()

Advance ledger time by the given interval.
This is only supported in Daml Studio and `daml test` as well as when running over the gRPC API against a ledger in static time mode. Note that this is not an atomic operation over the gRPC API so no other clients should try to change time while this is running.
Note that the ledger time service does not support going backwards in time. However, you can go back in time in Daml Studio.

allocateParty : HasCallStack => Text -> *Script* Party
Allocate a party with the given display name using the party management service.

allocatePartyWithHint : HasCallStack => Text -> *PartyIdHint* -> *Script* Party
Allocate a party with the given display name and id hint using the party management service.

allocatePartyOn : Text -> *ParticipantName* -> *Script* Party
Allocate a party with the given display name on the specified participant using the party management service.

allocatePartyWithHintOn : Text -> *PartyIdHint* -> *ParticipantName* -> *Script* Party
Allocate a party with the given display name and id hint on the specified participant using the party management service.

listKnownParties : HasCallStack => *Script* [*PartyDetails*]
List the parties known to the default participant.

listKnownPartiesOn : HasCallStack => *ParticipantName* -> *Script* [*PartyDetails*]
List the parties known to the given participant.

sleep : HasCallStack => RelTime -> *Script* ()
Sleep for the given duration.
This is primarily useful in tests where you repeatedly call `query` until a certain state is reached.
Note that this will sleep for the same duration in both wall clock and static time mode.

submitMulti : HasCallStack => [Party] -> [Party] -> *Commands* a -> *Script* a
`submitMulti actAs readAs cmds` submits `cmds` as a single transaction authorized by `actAs`. Fetched contracts must be visible to at least one party in the union of actAs and readAs.

submitMultiMustFail : HasCallStack => [Party] -> [Party] -> *Commands* a -> *Script* ()
`submitMultiMustFail actAs readAs cmds` behaves like `submitMulti actAs readAs cmds` but fails when `submitMulti` succeeds and the other way around.

createCmd : (Template t, HasAgreement t) => t -> *Commands* (ContractId t)
Create a contract of the given template.

exerciseCmd : Choice t c r => ContractId t -> c -> *Commands* r
Exercise a choice on the given contract.

exerciseByKeyCmd : (TemplateKey t k, Choice t c r) => k -> c -> *Commands* r
Exercise a choice on the contract with the given key.

createAndExerciseCmd : (Template t, Choice t c r, HasAgreement t) => t -> c -> *Commands* r
Create a contract and exercise a choice on it in the same transaction.

archiveCmd : Choice t Archive () => ContractId t -> *Commands* ()
Archive the given contract.
`archiveCmd cid` is equivalent to `exerciseCmd cid Archive`.

script : *Script* a -> *Script* a
Convenience helper to declare you are writing a Script.

This is only useful for readability and to improve type inference. Any expression of type `Script a` is a valid script regardless of whether it is implemented using `script` or not.

userIdToText : *UserId* -> Text
Extract the name-text from a user identitifer.

validateUserId : HasCallStack => Text -> *Script UserId*
Construct a user identifer from text. May throw InvalidUserId.

createUser : HasCallStack => *User* -> [*UserRight*] -> *Script* ()
Create a user with the given rights. May throw UserAlreadyExists.

createUserOn : HasCallStack => *User* -> [*UserRight*] -> *ParticipantName* -> *Script* ()
Create a user with the given rights on the given participant. May throw UserAlreadyExists.

getUser : HasCallStack => *UserId* -> *Script User*
Fetch a user record by user id. May throw UserNotFound.

getUserOn : HasCallStack => *UserId* -> *ParticipantName* -> *Script User*
Fetch a user record by user id from the given participant. May throw UserNotFound.

listAllUsers : *Script* [*User*]
List all users. This function may make multiple calls to underlying paginated ledger API.

listAllUsersOn : *ParticipantName* -> *Script* [*User*]
List all users on the given participant. This function may make multiple calls to underlying paginated ledger API.

grantUserRights : HasCallStack => *UserId* -> [*UserRight*] -> *Script* [*UserRight*]
Grant rights to a user. Returns the rights that have been newly granted. May throw UserNotFound.

grantUserRightsOn : HasCallStack => *UserId* -> [*UserRight*] -> *ParticipantName* -> *Script* [*UserRight*]
Grant rights to a user on the given participant. Returns the rights that have been newly granted. May throw UserNotFound.

revokeUserRights : HasCallStack => *UserId* -> [*UserRight*] -> *Script* [*UserRight*]
Revoke rights for a user. Returns the revoked rights. May throw UserNotFound.

revokeUserRightsOn : HasCallStack => *UserId* -> [*UserRight*] -> *ParticipantName* -> *Script* [*UserRight*]
Revoke rights for a user on the given participant. Returns the revoked rights. May throw UserNotFound.

deleteUser : HasCallStack => *UserId* -> *Script* ()
Delete a user. May throw UserNotFound.

deleteUserOn : HasCallStack => *UserId* -> *ParticipantName* -> *Script* ()
Delete a user on the given participant. May throw UserNotFound.

listUserRights : HasCallStack => *UserId* -> *Script* [*UserRight*]
List the rights of a user. May throw UserNotFound.

listUserRightsOn : HasCallStack => *UserId* -> *ParticipantName* -> *Script* [*UserRight*]
List the rights of a user on the given participant. May throw UserNotFound.

submitUser : HasCallStack => *UserId* -> *Commands* a -> *Script* a
Submit the commands with the actAs and readAs claims granted to a user. May throw UserNotFound.

submitUserOn : HasCallStack => *UserId* -> *ParticipantName* -> *Commands* a -> *Script* a

Submit the commands with the actAs and readAs claims granted to the user on the given participant. May throw UserNotFound.

1.43.5 Daml Trigger Library

The Daml Trigger library defines the API used to declare a Daml trigger. See *Daml Triggers - Off-Ledger Automation in Daml*:: for more information on Daml triggers.

1.43.5.1 Daml.Trigger

Typeclasses

class *ActionTriggerAny* m **where**

Features possible in `initialize`, `updateState`, and `rule`.

queryContractId : Template a => ContractId a -> m (Optional a)
Find the contract with the given `id` in the ACS, if present.

getReadAs : m [Party]

getActAs : m Party

instance *ActionTriggerAny* (*TriggerA* s)

instance *ActionTriggerAny TriggerInitializeA*

instance *ActionTriggerAny* (*TriggerUpdateA* s)

class *ActionTriggerAny* m => *ActionTriggerUpdate* m **where**

Features possible in `updateState` and `rule`.

getCommandsInFlight : m (Map *CommandId* [*Command*])
Retrieve command submissions made by this trigger that have not yet completed. If the trigger has restarted, it will not contain commands from before the restart; therefore, this should be treated as an optimization rather than an absolute authority on ledger state.

instance *ActionTriggerUpdate* (*TriggerA* s)

instance *ActionTriggerUpdate* (*TriggerUpdateA* s)

Data Types

data *Trigger* s

This is the type of your trigger. `s` is the user-defined state type which you can often leave at `()`.

Trigger

Field	Type	Description
initialize	*TriggerInitializeA* s	Initialize the user-defined state based on the ACS.
updateState	*Message* -> *TriggerUpdateA* s ()	Update the user-defined state based on a transaction or completion message. It can manipulate the state with `get`, `put`, and `modify`, or query the ACS with `query`.
rule	Party -> *TriggerA* s ()	The rule defines the main logic of your trigger. It can send commands to the ledger using `emitCommands` to change the ACS. The rule depends on the following arguments: * The party your trigger is running as. * The user-defined state. and can retrieve other data with functions in `TriggerA`: * The current state of the ACS. * The current time (UTC in wall-clock mode, Unix epoch in static mode) * The commands in flight.
registeredTemplates	*RegisteredTemplates*	The templates the trigger will receive events for.
heartbeat	Optional RelTime	Send a heartbeat message at the given interval.

instance HasField "heartbeat" (*Trigger* s) (Optional RelTime)

instance HasField "initialize" (*Trigger* s) (*TriggerInitializeA* s)

instance HasField "registeredTemplates" (*Trigger* s) *RegisteredTemplates*

instance HasField "rule" (*Trigger* s) (Party -> *TriggerA* s ())

instance HasField "updateState" (*Trigger* s) (*Message* -> *TriggerUpdateA* s ())

data *TriggerA* s a

TriggerA is the type used in the `rule` of a Daml trigger. Its main feature is that you can call `emitCommands` to send commands to the ledger.

instance *ActionTriggerAny* (*TriggerA* s)

instance *ActionTriggerUpdate* (*TriggerA* s)

instance Functor (*TriggerA* s)

instance ActionState s (*TriggerA* s)

instance HasTime (*TriggerA* s)

instance Action (*TriggerA* s)

instance Applicative (*TriggerA* s)

instance HasField "rule" (*Trigger* s) (Party -> *TriggerA* s ())

instance HasField "runTriggerA" (*TriggerA* s a) (ACS -> *TriggerRule* (TriggerAState s) a)

data *TriggerInitializeA* a

TriggerInitializeA is the type used in the `initialize` of a Daml trigger. It can query, but not emit commands or update the state.

instance *ActionTriggerAny TriggerInitializeA*

instance Functor *TriggerInitializeA*

instance Action *TriggerInitializeA*

instance Applicative *TriggerInitializeA*

instance HasField "initialize" (*Trigger* s) (*TriggerInitializeA* s)

instance HasField "runTriggerInitializeA" (*TriggerInitializeA* a) (TriggerInitState -> a)

data *TriggerUpdateA* s a

TriggerUpdateA is the type used in the `updateState` of a Daml trigger. It has similar actions in common with `TriggerA`, but cannot use `emitCommands` or `getTime`.

instance *ActionTriggerAny* (*TriggerUpdateA* s)

instance *ActionTriggerUpdate* (*TriggerUpdateA* s)

instance Functor (*TriggerUpdateA* s)

instance ActionState s (*TriggerUpdateA* s)

instance Action (*TriggerUpdateA* s)

instance Applicative (*TriggerUpdateA* s)

instance HasField "runTriggerUpdateA" (*TriggerUpdateA* s a) (TriggerUpdateState -> State s a)

instance HasField "updateState" (*Trigger* s) (*Message* -> *TriggerUpdateA* s ())

Functions

query : (Template a, *ActionTriggerAny* m) => m [(ContractId a, a)]
Extract the contracts of a given template from the ACS.

queryFilter : (Functor m, Template a, *ActionTriggerAny* m) => (a -> Bool) -> m [(ContractId a, a)]
Extract the contracts of a given template from the ACS and filter to those that match the predicate.

queryContractKey : (Template a, HasKey a k, Eq k, *ActionTriggerAny* m, Functor m) => k -> m (Optional (ContractId a, a))
Find the contract with the given `key` in the ACS, if present.

emitCommands : [*Command*] -> [*AnyContractId*] -> *TriggerA* s *CommandId*
Send a transaction consisting of the given commands to the ledger. The second argument can be used to mark a list of contract ids as pending. These contracts will automatically be filtered from getContracts until we either get the corresponding transaction event for this command or a failing completion.

emitCommandsV2 : [*Command*] -> [*AnyContractId*] -> *TriggerA* s (Optional *CommandId*)

dedupCreate : (Eq t, Template t) => t -> *TriggerA* s ()
Create the template if it's not already in the list of commands in flight (it will still be created if it is in the ACS).
Note that this will send the create as a single-command transaction. If you need to send multiple commands in one transaction, use `emitCommands` with `createCmd` and handle filtering yourself.

dedupCreateAndExercise : (Eq t, Eq c, Template t, Choice t c r) => t -> c -> *TriggerA* s ()
Create the template and exercise a choice on it if it's not already in the list of commands in flight (it will still be created if it is in the ACS).
Note that this will send the create and exercise as a single-command transaction. If you need to send multiple commands in one transaction, use `emitCommands` with `createAndExerciseCmd` and handle filtering yourself.

dedupExercise : (Eq c, Choice t c r) => ContractId t -> c -> *TriggerA* s ()
Exercise the choice on the given contract if it is not already in flight.
Note that this will send the exercise as a single-command transaction. If you need to send multiple commands in one transaction, use `emitCommands` with `exerciseCmd` and handle filtering yourself.
If you are calling a consuming choice, you might be better off by using `emitCommands` and adding the contract id to the pending set.

dedupExerciseByKey : (Eq c, Eq k, Choice t c r, TemplateKey t k) => k -> c -> *TriggerA* s ()
Exercise the choice on the given contract if it is not already in flight.
Note that this will send the exercise as a single-command transaction. If you need to send multiple commands in one transaction, use `emitCommands` with `exerciseCmd` and handle filtering yourself.

runTrigger : *Trigger* s -> *BatchTrigger* (TriggerState s)
Transform the high-level trigger type into the batching trigger from `Daml.Trigger.LowLevel`.

1.43.5.2 Daml.Trigger.Assert

Data Types

data *ACSBuilder*

Used to construct an 'ACS' for 'testRule'.

instance Monoid *ACSBuilder*

instance Semigroup *ACSBuilder*

Functions

toACS : (Template t, HasAgreement t) => ContractId t -> *ACSBuilder*
Include the given contract in the 'ACS'. Note that the `ContractId` must point to an active contract.

testRule : *Trigger* s -> Party -> [Party] -> *ACSBuilder* -> Map *CommandId* [*Command*] -> s -> Script (s, [*Commands*])
Execute a trigger's rule once in a scenario.

flattenCommands : [*Commands*] -> [*Command*]
Drop 'CommandId's and extract all 'Command's.

assertCreateCmd : (Template t, HasAgreement t, CanAbort m) => [*Command*] -> (t -> Either Text ()) -> m ()
Check that at least one command is a create command whose payload fulfills the given assertions.

assertExerciseCmd : (Template t, HasAgreement t, Choice t c r, CanAbort m) => [*Command*] -> ((ContractId t, c) -> Either Text ()) -> m ()
Check that at least one command is an exercise command whose contract id and choice argument fulfill the given assertions.

assertExerciseByKeyCmd : (TemplateKey t k, Choice t c r, CanAbort m) => [*Command*] -> ((k, c) -> Either Text ()) -> m ()
Check that at least one command is an exercise by key command whose key and choice argument fulfill the given assertions.

1.43.5.3 Daml.Trigger.LowLevel

Typeclasses

class HasTime m => *ActionTrigger* m **where**

Low-level trigger actions.

liftTF : TriggerF a -> m a

instance *ActionTrigger* (*TriggerRule* s)

instance *ActionTrigger TriggerSetup*

Data Types

data *ActiveContracts*

ActiveContracts

Field	Type	Description
activeContracts	[*Created*]	

instance HasField "acs" *TriggerSetupArguments ActiveContracts*

instance HasField "activeContracts" *ActiveContracts* [*Created*]

instance HasField "initialState" (*Trigger* s) (Party -> [Party] -> *ActiveContracts* -> *TriggerSetup* s)

data *AnyContractId*

This type represents the contract id of an unknown template. You can use `fromAnyContractId` to check which template it corresponds to.

instance Eq *AnyContractId*

instance Ord *AnyContractId*

instance Show *AnyContractId*

instance HasField "activeContracts" ACS (Map TemplateTypeRep (Map *AnyContractId* AnyTemplate))

instance HasField "contractId" *AnyContractId* (ContractId ())

instance HasField "contractId" *Archived AnyContractId*

instance HasField "contractId" *Command AnyContractId*

instance HasField "contractId" *Created AnyContractId*

instance HasField "pendingContracts" ACS (Map *CommandId* [*AnyContractId*])

instance HasField "pendingContracts" (TriggerAState s) (Map *CommandId* [*AnyContractId*])

instance HasField "templateId" *AnyContractId* TemplateTypeRep

data *Archived*

The data in an `Archived` event.

Archived

Field	Type	Description
eventId	*EventId*	
contractId	*AnyContrac-tId*	

instance Eq *Archived*

instance Show *Archived*

instance HasField "contractId" *Archived AnyContractId*

instance HasField "eventId" *Archived EventId*

data *BatchTrigger* s

Batching trigger is (approximately) a left-fold over `Message` with an accumulator of type `s`.

BatchTrigger

Field	Type	Description
initialState	*TriggerSetupArguments* -> *TriggerSetup* s	
update	[*Message*] -> *TriggerRule* s ()	
registeredTemplates	*RegisteredTemplates*	
heartbeat	Optional RelTime	

instance HasField "heartbeat" (*BatchTrigger* s) (Optional RelTime)

instance HasField "initialState" (*BatchTrigger* s) (*TriggerSetupArguments* -> *TriggerSetup* s)

instance HasField "registeredTemplates" (*BatchTrigger* s) *RegisteredTemplates*

instance HasField "update" (*BatchTrigger* s) ([*Message*] -> *TriggerRule* s ())

data *Command*

A ledger API command. To construct a command use `createCmd` and `exerciseCmd`.

CreateCommand

Field	Type	Description
templateArg	AnyTemplate	

ExerciseCommand

Field	Type	Description
contractId	*AnyContractId*	
choiceArg	AnyChoice	

CreateAndExerciseCommand

Field	Type	Description
templateArg	AnyTemplate	
choiceArg	AnyChoice	

ExerciseByKeyCommand

Field	Type	Description
tplTypeRep	Template-TypeRep	
contractKey	AnyCon-tractKey	
choiceArg	AnyChoice	

instance HasField "choiceArg" *Command* AnyChoice

instance HasField "commands" *Commands* [*Command*]

instance HasField "commandsInFlight" (TriggerAState s) (Map *CommandId* [*Command*])

instance HasField "commandsInFlight" (TriggerState s) (Map *CommandId* [*Command*])

instance HasField "commandsInFlight" TriggerUpdateState (Map *CommandId* [*Command*])

instance HasField "contractId" *Command AnyContractId*

instance HasField "contractKey" *Command* AnyContractKey

instance HasField "templateArg" *Command* AnyTemplate

instance HasField "tplTypeRep" *Command* TemplateTypeRep

data *CommandId*

CommandId Text

instance Eq *CommandId*

instance Ord *CommandId*

instance Show *CommandId*

instance HasField "commandId" *Commands CommandId*

instance HasField "commandId" *Completion CommandId*

instance HasField "commandId" *Transaction* (Optional *CommandId*)

instance HasField "commandsInFlight" (TriggerAState s) (Map *CommandId* [*Command*])

instance HasField "commandsInFlight" (TriggerState s) (Map *CommandId* [*Command*])

instance HasField "commandsInFlight" TriggerUpdateState (Map *CommandId* [*Command*])

instance HasField "pendingContracts" ACS (Map *CommandId* [*AnyContractId*])

instance HasField "pendingContracts" (TriggerAState s) (Map *CommandId* [*AnyContractId*])

data *Commands*

A set of commands that are submitted as a single transaction.

Commands

Field	Type	Description
commandId	*CommandId*	
commands	[*Command*]	

instance HasField "commandId" *Commands CommandId*

instance HasField "commands" *Commands* [*Command*]

data *Completion*

A completion message. Note that you will only get completions for commands emitted from the trigger. Contrary to the ledger API completion stream, this also includes synchronous failures.

Completion

Field	Type	Description
commandId	*CommandId*	
status	*CompletionStatus*	

instance Show *Completion*

instance HasField "commandId" *Completion CommandId*

instance HasField "status" *Completion CompletionStatus*

data *CompletionStatus*

Failed

Field	Type	Description
status	Int	
message	Text	

Succeeded

Field	Type	Description
transactionId	*TransactionId*	

instance Show *CompletionStatus*

instance HasField "message" *CompletionStatus* Text

instance HasField "status" *Completion CompletionStatus*

instance HasField "status" *CompletionStatus* Int

instance HasField "transactionId" *CompletionStatus TransactionId*

data *Created*

The data in a `Created` event.

Created

Field	Type	Description
eventId	*EventId*	
contractId	*AnyContractId*	
argument	Optional AnyTemplate	
views	[InterfaceView]	

instance HasField "activeContracts" *ActiveContracts* [*Created*]

instance HasField "argument" *Created* (Optional AnyTemplate)

instance HasField "contractId" *Created AnyContractId*

instance HasField "eventId" *Created EventId*

instance HasField "views" *Created* [InterfaceView]

data *Event*

An event in a transaction. This definition should be kept consistent with the object `EventVariant` defined in triggers/runner/src/main/scala/com/digitalasset/daml/lf/engine/trigger/Converter.scala

CreatedEvent Created

ArchivedEvent Archived

instance HasField "events" *Transaction* [*Event*]

data *EventId*

EventId Text

instance Eq *EventId*

instance Show *EventId*

instance HasField "eventId" *Archived EventId*

instance HasField "eventId" *Created EventId*

data *Message*

Either a transaction or a completion. This definition should be kept consistent with the object `MessageVariant` defined in triggers/runner/src/main/scala/com/digitalasset/daml/lf/engine/trigger/Converter.scala

MTransaction Transaction

MCompletion Completion

MHeartbeat

instance HasField "update" (*BatchTrigger* s) ([*Message*] -> *TriggerRule* s ())

instance HasField "update" (*Trigger* s) (*Message* -> *TriggerRule* s ())

instance HasField "updateState" (*Trigger* s) (*Message* -> *TriggerUpdateA* s ())

data *RegisteredTemplates*

AllInDar

Listen to events for all templates in the given DAR.

RegisteredTemplates [RegisteredTemplate]

instance HasField "registeredTemplates" (*BatchTrigger* s) *RegisteredTemplates*

instance HasField "registeredTemplates" (*Trigger* s) *RegisteredTemplates*

instance HasField "registeredTemplates" (*Trigger* s) *RegisteredTemplates*

data *Transaction*

Transaction

Field	Type	Description
transactionId	*TransactionId*	
commandId	Optional *CommandId*	
events	[*Event*]	

instance HasField "commandId" *Transaction* (Optional *CommandId*)

instance HasField "events" *Transaction* [*Event*]

instance HasField "transactionId" *Transaction TransactionId*

data *TransactionId*

TransactionId Text

instance Eq *TransactionId*

instance Show *TransactionId*

instance HasField "transactionId" *CompletionStatus TransactionId*

instance HasField "transactionId" *Transaction TransactionId*

data *Trigger* s

Trigger

Field	Type	Description
initialState	Party -> [Party] -> *ActiveContracts* -> *TriggerSetup* s	
update	*Message* -> *TriggerRule* s ()	
registeredTemplates	*RegisteredTemplates*	
heartbeat	Optional RelTime	

instance HasField "heartbeat" (*Trigger* s) (Optional RelTime)

instance HasField "initialState" (*Trigger* s) (Party -> [Party] -> *ActiveContracts* -> *TriggerSetup* s)

instance HasField "registeredTemplates" (*Trigger* s) *RegisteredTemplates*

instance HasField "update" (*Trigger* s) (*Message* -> *TriggerRule* s ())

data *TriggerConfig*

TriggerConfig

Field	Type	Description
maxInFlightCommands	Int	maximum number of commands that should be allowed to be in-flight at any point in time. Exceeding this value will eventually lead to the trigger run raising an InFlightCommandOverflowException exception.
maxActiveContracts	Int	maximum number of active contracts that we will allow to be stored Exceeding this value will lead to the trigger runner raising an ACSOverflowException exception.

instance HasField "config" (TriggerAState s) *TriggerConfig*

instance HasField "config" (TriggerState s) *TriggerConfig*

instance HasField "config" *TriggerSetupArguments TriggerConfig*

instance HasField "maxActiveContracts" *TriggerConfig* Int

instance HasField "maxInFlightCommands" *TriggerConfig* Int

data *TriggerRule* s a

TriggerRule

Field	Type	Description
runTriggerRule	StateT s (Free TriggerF) a	

instance ActionTrigger (*TriggerRule* s)

instance Functor (*TriggerRule* s)

instance ActionState s (*TriggerRule* s)

instance HasTime (*TriggerRule* s)

instance Action (*TriggerRule* s)

instance Applicative (*TriggerRule* s)

instance HasField "runTriggerA" (*TriggerA* s a) (ACS -> *TriggerRule* (TriggerAState s) a)

instance HasField "runTriggerRule" (*TriggerRule* s a) (StateT s (Free TriggerF) a)

instance HasField "update" (*BatchTrigger* s) ([*Message*] -> *TriggerRule* s ())

instance HasField "update" (*Trigger* s) (*Message* -> *TriggerRule* s ())

data *TriggerSetup* a

TriggerSetup

Field	Type	Description
runTriggerSetup	Free TriggerF a	

instance ActionTrigger *TriggerSetup*

instance Functor *TriggerSetup*

instance HasTime *TriggerSetup*

instance Action *TriggerSetup*

instance Applicative *TriggerSetup*

instance HasField "initialState" (*BatchTrigger* s) (*TriggerSetupArguments* -> *TriggerSetup* s)

instance HasField "initialState" (*Trigger* s) (Party -> [Party] -> *ActiveContracts* -> *TriggerSetup* s)

instance HasField "runTriggerSetup" (*TriggerSetup* a) (Free TriggerF a)

data *TriggerSetupArguments*

TriggerSetupArguments

Field	Type	Description
actAs	Party	
readAs	[Party]	
acs	*ActiveContracts*	
config	*TriggerConfig*	

instance HasField "acs" *TriggerSetupArguments ActiveContracts*

instance HasField "actAs" *TriggerSetupArguments* Party

instance HasField "config" *TriggerSetupArguments TriggerConfig*

instance HasField "initialState" (*BatchTrigger* s) (*TriggerSetupArguments* -> *TriggerSetup* s)

instance HasField "readAs" *TriggerSetupArguments* [Party]

Functions

toAnyContractId : Template t => ContractId t -> *AnyContractId*
Wrap a `ContractId t` in `AnyContractId`.

fromAnyContractId : Template t => *AnyContractId* -> Optional (ContractId t)
Check if a `AnyContractId` corresponds to the given template or return `None` otherwise.

fromCreated : Template t => *Created* -> Optional (*EventId*, ContractId t, t)
Check if a `Created` event corresponds to the given template.

fromArchived : Template t => *Archived* -> Optional (*EventId*, ContractId t)
Check if an `Archived` event corresponds to the given template.

registeredTemplate : Template t => RegisteredTemplate

createCmd : Template t => t -> *Command*
Create a contract of the given template.

exerciseCmd : Choice t c r => ContractId t -> c -> *Command*
Exercise the given choice.

createAndExerciseCmd : (Template t, Choice t c r) => t -> c -> *Command*
Create a contract of the given template and immediately exercise the given choice on it.

exerciseByKeyCmd : (Choice t c r, TemplateKey t k) => k -> c -> *Command*

fromCreate : Template t => *Command* -> Optional t
Check if the command corresponds to a create command for the given template.

fromCreateAndExercise : (Template t, Choice t c r) => *Command* -> Optional (t, c)
Check if the command corresponds to a create and exercise command for the given template.

fromExercise : Choice t c r => *Command* -> Optional (ContractId t, c)
Check if the command corresponds to an exercise command for the given template.

fromExerciseByKey : (Choice t c r, TemplateKey t k) => *Command* -> Optional (k, c)
Check if the command corresponds to an exercise by key command for the given template.

execStateT : Functor m => StateT s m a -> s -> m s

zoom : Functor m => (t -> s) -> (t -> s -> t) -> StateT s m a -> StateT t m a

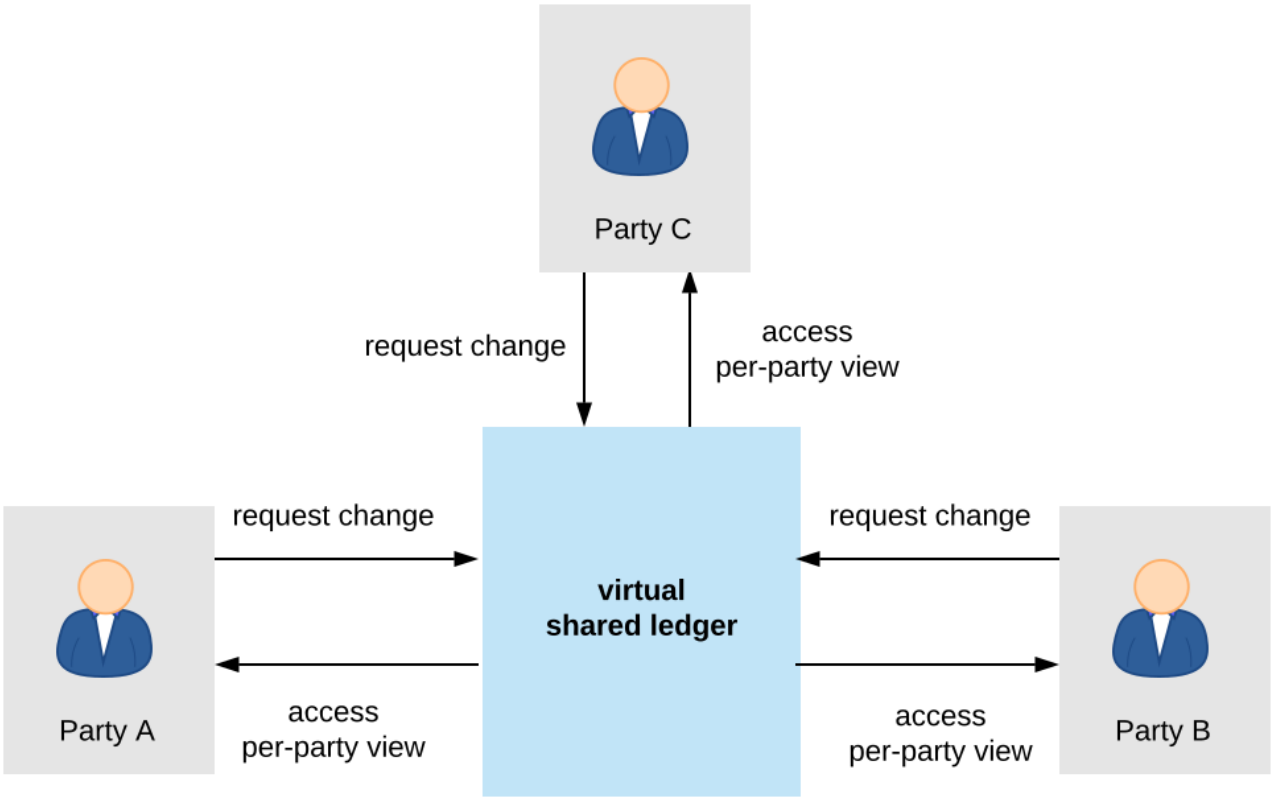
simulateRule : *TriggerRule* s a -> Time -> s -> (s, [*Commands*], a)
Run a rule without running it. May lose information from the rule; meant for testing purposes only.

submitCommands : *ActionTrigger* m => [*Command*] -> m *CommandId*

1.44 Daml Ledger References

1.44.1 Daml Ledger Model

Daml Ledgers enable multi-party workflows by providing parties with a virtual *shared ledger*, which encodes the current state of their shared contracts, written in Daml. At a high level, the interactions are visualized as follows:



The Daml ledger model defines:

1. what the ledger looks like - the structure of Daml ledgers
2. who can request which changes - the integrity model for Daml ledgers
3. who sees which changes and data - the privacy model for Daml ledgers

The below sections review these concepts of the ledger model in turn. They also briefly describe the link between Daml and the model.

1.44.1.1 Structure

This section looks at the structure of a Daml ledger and the associated ledger changes. The basic building blocks of changes are *actions*, which get grouped into *transactions*.

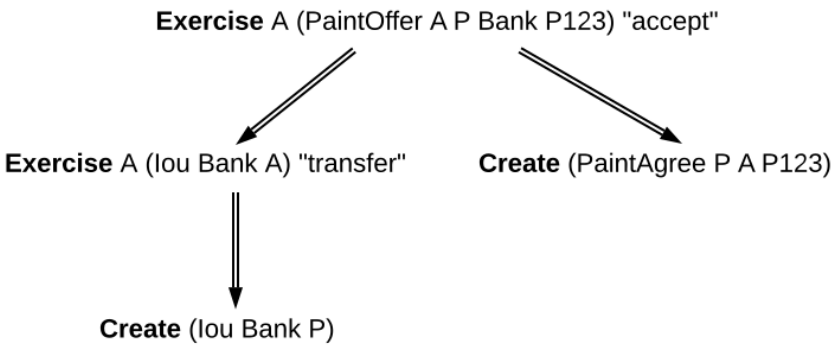
Actions and Transactions

One of the main features of the Daml ledger model is a *hierarchical action structure*.

This structure is illustrated below on a toy example of a multi-party interaction. Alice (*A*) gets some digital cash, in the form of an I-Owe-You (IOU for short) from a bank, and she needs her house painted. She gets an offer from a painter (*P*) with reference number *P123* to paint her house in exchange for this IOU. Lastly, *A* accepts the offer, transferring the money and signing a contract with *P*, whereby he is promising to paint her house.

This acceptance can be viewed as *A exercising* her right to accept the offer. Her acceptance has two consequences. First, *A* transfers her IOU, that is, *exercises* her right to transfer the IOU, after which a new IOU for *P* is *created*. Second, a new contract is *created* that requires *P* to paint *A*'s house.

Thus, the acceptance in this example is reduced to two types of actions: (1) creating contracts, and (2) exercising rights on them. These are also the two main kinds of actions in the Daml ledger model. The visual notation below records the relations between the actions during the above acceptance.



Formally, an **action** is one of the following:

1. a **Create** action on a contract, which records the creation of the contract
2. an **Exercise** action on a contract, which records that one or more parties have exercised a right they have on the contract, and which also contains:
    1. An associated set of parties called **actors**. These are the parties who perform the action.
    2. An exercise **kind**, which is either **consuming** or **non-consuming**. Once consumed, a contract cannot be used again (for example, Alice should not be able to accept the painter's offer twice). Contracts exercised in a non-consuming fashion can be reused.
    3. A list of **consequences**, which are themselves actions. Note that the consequences, as well as the kind and the actors, are considered a part of the exercise action itself. This nesting of actions within other actions through consequences of exercises gives rise to the hierarchical structure. The exercise action is the **parent action** of its consequences.
3. a **Fetch** action on a contract, which demonstrates that the contract exists and is active at the time of fetching. The action also contains **actors**, the parties who fetch the contract. A **Fetch** behaves like a non-consuming exercise with no consequences, and can be repeated.
4. a **Key assertion**, which records the assertion that the given *contract key* is **not** assigned to any unconsumed contract on the ledger.

An **Exercise** or a **Fetch** action on a contract is said to **use** the contract. Moreover, a consuming **Exercise** is said to **consume** (or **archive**) its contract.

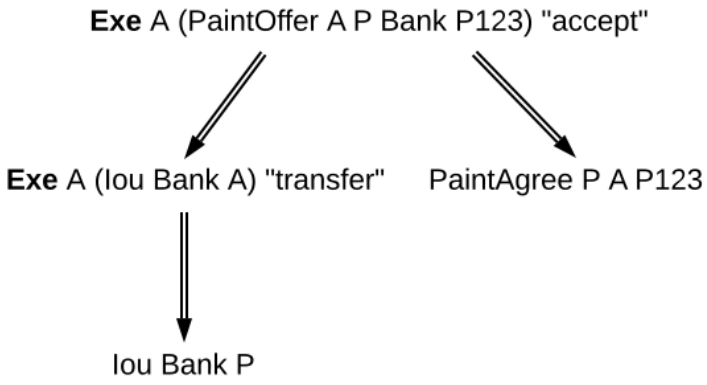
The following EBNF-like grammar summarizes the structure of actions and transactions. Here, `s | t` represents the choice between *s* and *t*, `s t` represents *s* followed by *t*, and `s*` represents the repetition of *s* zero or more times. The terminal 'contract' denotes the underlying type of contracts, and the terminal 'party' the underlying type of parties.

```
Action       ::= 'Create' contract
               | 'Exercise' party* contract Kind Transaction
               | 'Fetch' party* contract
               | 'NoSuchKey' key
Transaction  ::= Action*
Kind         ::= 'Consuming' | 'NonConsuming'
```

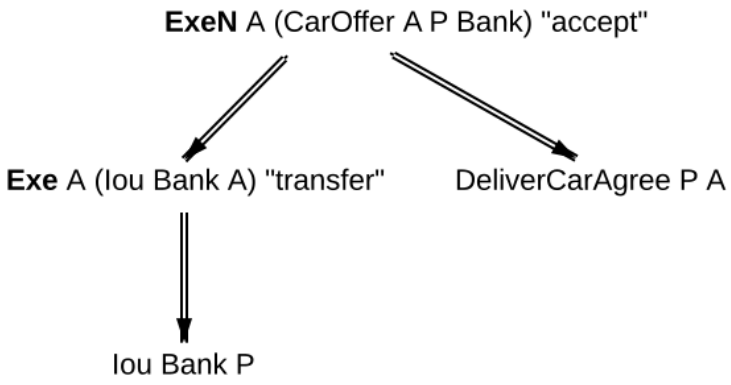
The visual notation presented earlier captures actions precisely with conventions that:

1. **Exercise** denotes consuming, **ExerciseN** non-consuming exercises, and **Fetch** a fetch.
2. double arrows connect exercises to their consequences, if any.
3. the consequences are ordered left-to-right.
4. to aid intuitions, exercise actions are annotated with suggestive names like `accept` or `transfer`. Intuitively, these correspond to names of Daml choices, but they have no semantic meaning.

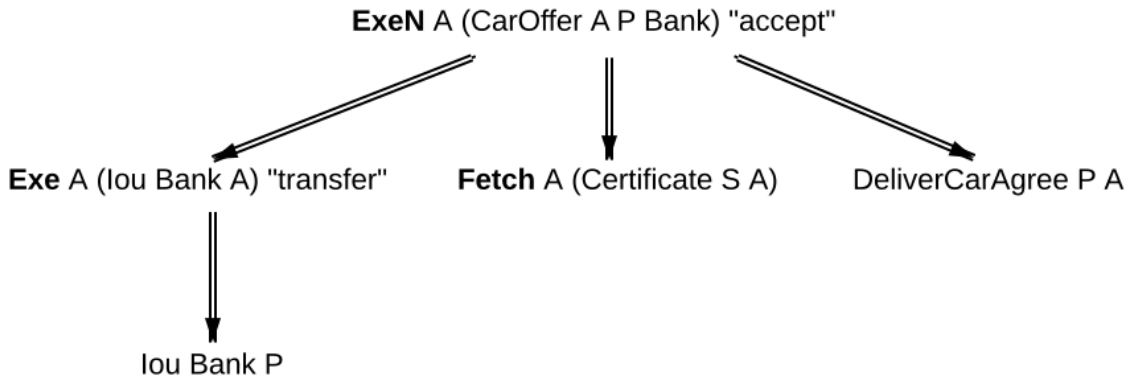
An alternative shorthand notation, shown below uses the abbreviations **Exe** and **ExeN** for exercises, and omits the **Create** labels on create actions.



To show an example of a non-consuming exercise, consider a different offer example with an easily replenishable subject. For example, if *P* was a car manufacturer, and *A* a car dealer, *P* could make an offer that could be accepted multiple times.



To see an example of a fetch, we can extend this example to the case where *P* produces exclusive cars and allows only certified dealers to sell them. Thus, when accepting the offer, *A* has to additionally show a valid quality certificate issued by some standards body *S*.



In the paint offer example, the underlying type of contracts consists of three sorts of contracts:

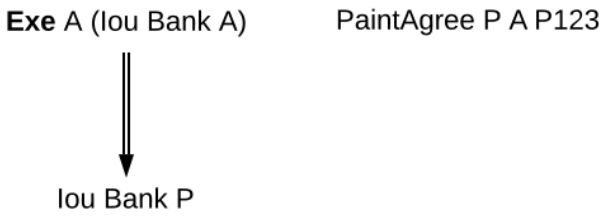
PaintOffer houseOwner painter obligor refNo Intuitively an offer (with a reference number) by which the painter proposes to the house owner to paint her house, in exchange for a single IOU token issued by the specified obligor.

PaintAgree painter houseOwner refNo Intuitively a contract whereby the painter agrees to paint the owner's house

Iou obligor owner An IOU token from an obligor to an owner (for simplicity, the token is of unit amount).

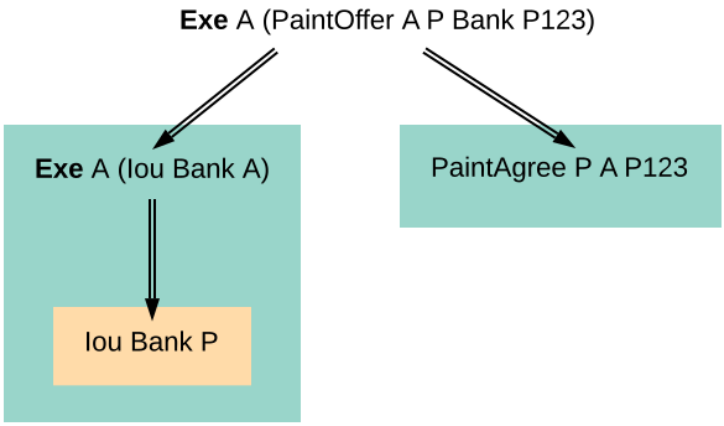
In practice, multiple IOU contracts can exist between the same *obligor* and *owner*, in which case each contract should have a unique identifier. However, in this section, each contract only appears once, allowing us to drop the notion of identifiers for simplicity reasons.

A **transaction** is a list of actions. Thus, the consequences of an exercise form a transaction. In the example, the consequences of Alice's exercise form the following transaction, where actions are again ordered left-to-right.

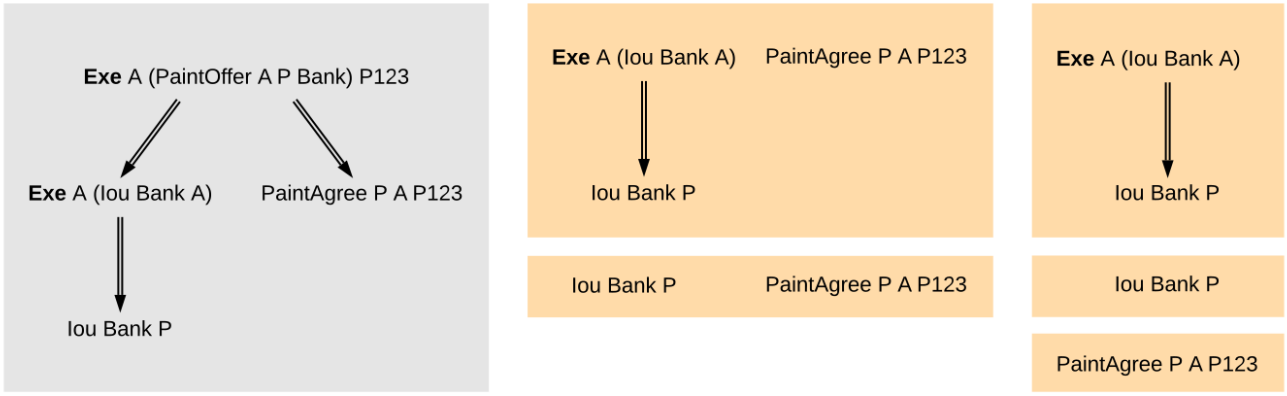


For an action *act*, its **proper subactions** are all actions in the consequences of *act*, together with all of their proper subactions. Additionally, *act* is a (non-proper) **subaction** of itself.

The subaction relation is visualized below. Both the green and yellow boxes are proper subactions of Alice's exercise on the paint offer. Additionally, the creation of *Iou Bank P* (yellow box) is also a proper subaction of the exercise on the *Iou Bank A*.



Similarly, a **subtransaction** of a transaction is either the transaction itself, or a **proper subtransaction**: a transaction obtained by removing at least one action, or replacing it by a subtransaction of its consequences. For example, given the transaction consisting of just one action, the paint offer acceptance, the image below shows all its proper non-empty subtransactions on the right (yellow boxes).



To illustrate *contract keys*, suppose that the contract key for a *PaintOffer* consists of the reference number and the painter. So Alice can refer to the *PaintOffer* by its key *(P, P123)*. To make this explicit, we use the notation *PaintOffer @P A &P123* for contracts, where @ and & mark the parts that belong to a key. (The difference between @ and & will be explained in the *integrity section*.) The ledger integrity

constraints in the next section ensure that there is always at most one active *PaintOffer* for a given key. So if the painter retracts its *PaintOffer* and later Alice tries to accept it, she can then record the absence with a *NoSuchKey (P, P123)* key assertion.

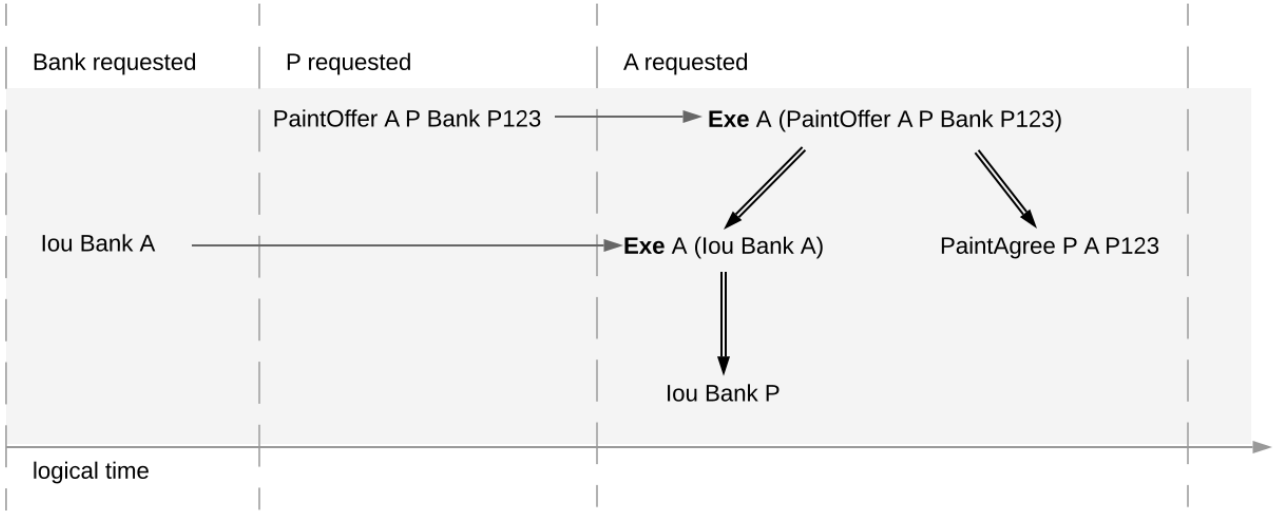
Ledgers

The transaction structure records the contents of the changes, but not *who requested them.* This information is added by the notion of a **commit**: a transaction paired with the parties that requested it, called the **requesters** of the commit. A commit may have one or more requesters. Given a commit *(p, tx)* with transaction $tx = act_1, \ldots, act_n$, every act_i is called a **top-level action** of the commit. A **ledger** is a sequence of commits. A top-level action of any ledger commit is also a top-level action of the ledger.

The following EBNF grammar summarizes the structure of commits and ledgers:

```
Commit    ::= party+ Transaction
Ledger    ::= Commit*
```

A Daml ledger thus represents the full history of all actions taken by parties.[1] Since the ledger is a sequence (of dependent actions), it induces an *order* on the commits in the ledger. Visually, a ledger can be represented as a sequence growing from left to right as time progresses. Below, dashed vertical lines mark the boundaries of commits, and each commit is annotated with its requester(s). Arrows link the create and exercise actions on the same contracts. These additional arrows highlight that the ledger forms a **transaction graph**. For example, the aforementioned house painting scenario is visually represented as follows.



The definitions presented here are all the ingredients required to *record* the interaction between parties in a Daml ledger. That is, they address the first question: what do changes and ledgers look like? . To answer the next question, who can request which changes , a precise definition is needed of which ledgers are permissible, and which are not. For example, the above paint offer ledger is intuitively permissible, while all of the following ledgers are not.

The next section discusses the criteria that rule out the above examples as invalid ledgers.

[1] Calling such a complete record ledger is standard in the distributed ledger technology community. In accounting terminology, this record is closer to a *journal* than to a ledger.

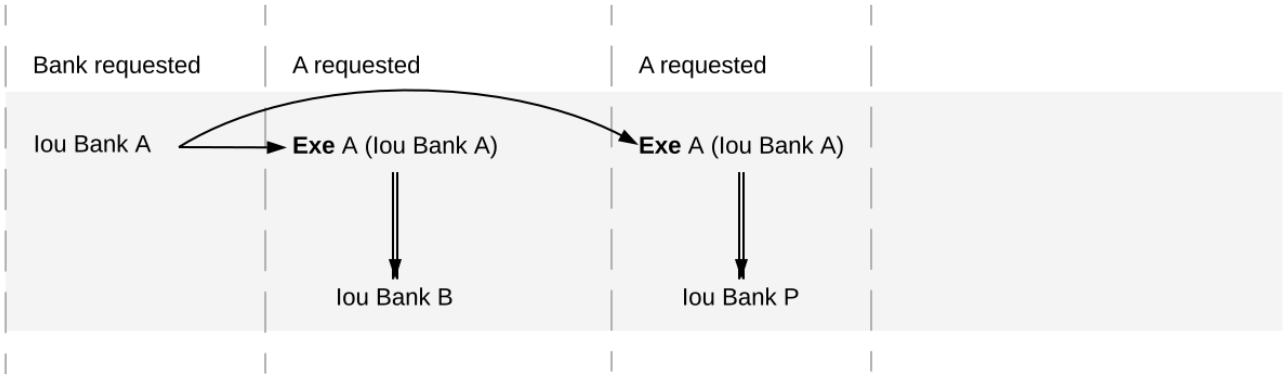


Fig. 19: Alice spending her IOU twice (double spend), once transferring it to *B* and once to *P*.

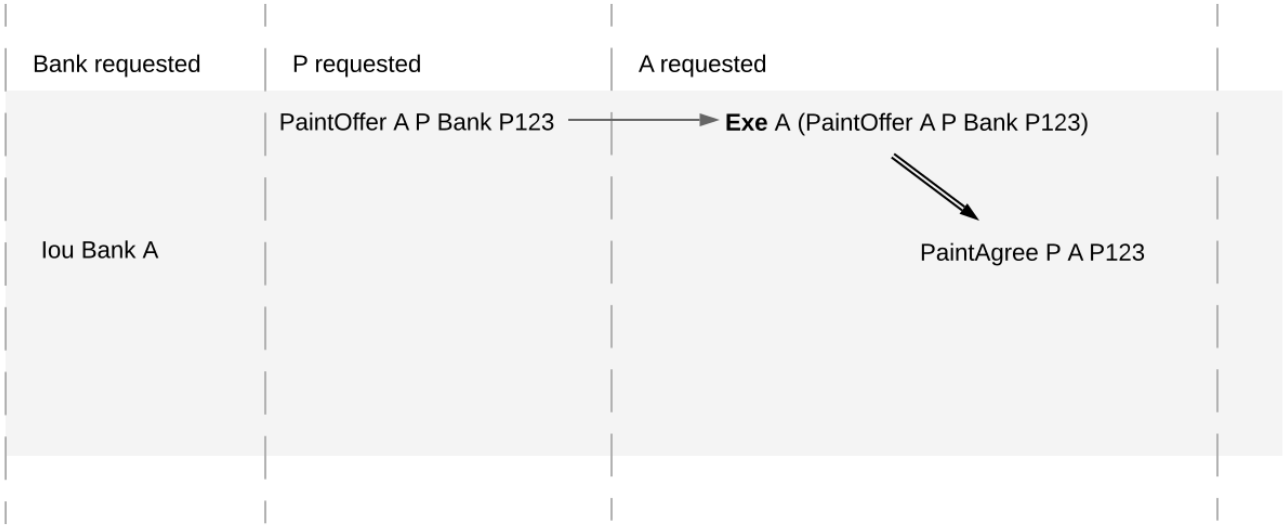


Fig. 20: Alice changing the offer's outcome by removing the transfer of the *Iou*.

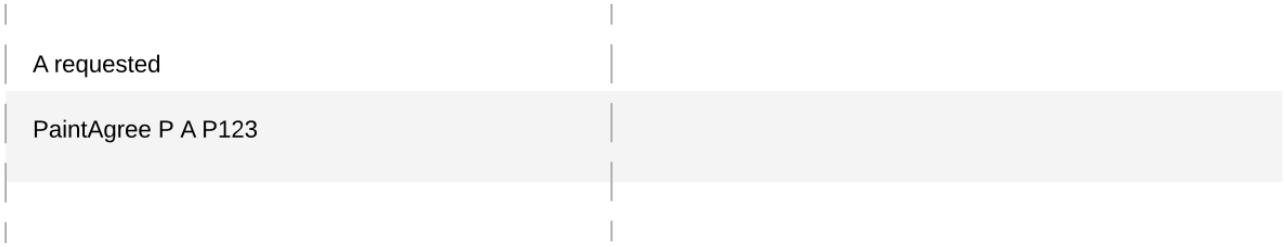


Fig. 21: An obligation imposed on the painter without his consent.

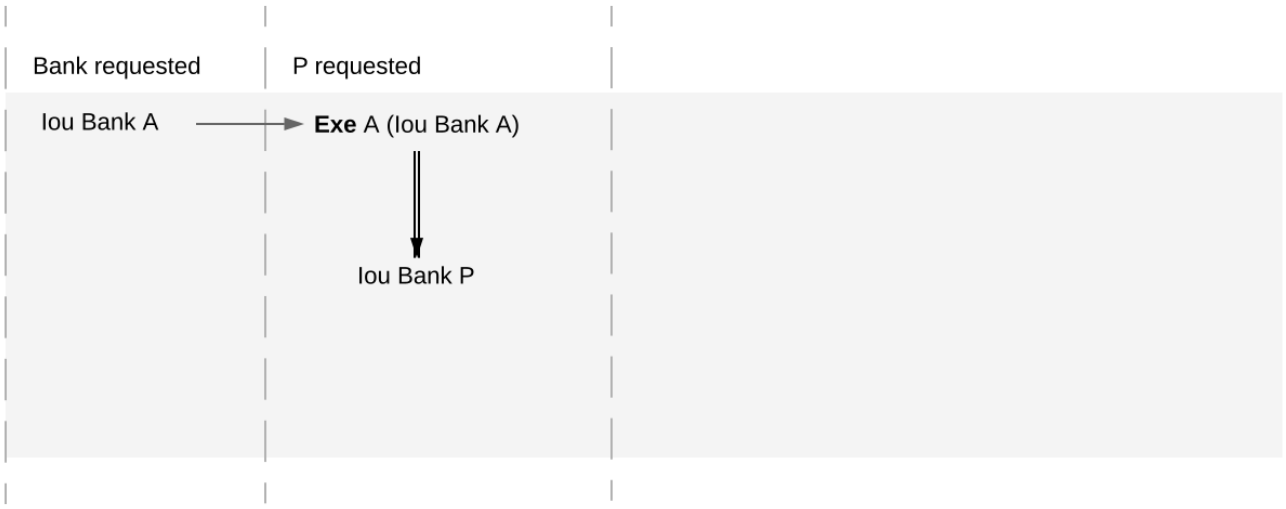


Fig. 22: Painter stealing Alice's IOU. Note that the ledger would be intuitively permissible if it was Alice performing the last commit.

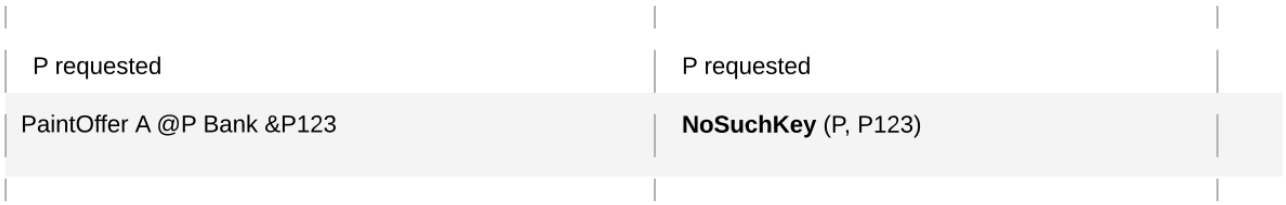


Fig. 23: Painter falsely claiming that there is no offer.

Fig. 24: Painter trying to create two different paint offers with the same reference number.

1.44.1.2 Integrity

This section addresses the question of who can request which changes.

Valid Ledgers

At the core is the concept of a *valid ledger*; changes are permissible if adding the corresponding commit to the ledger results in a valid ledger. **Valid ledgers** are those that fulfill three conditions:

Consistency Exercises and fetches on inactive contracts are not allowed, i.e. contracts that have not yet been created or have already been consumed by an exercise. A contract with a contract key can be created only if the key is not associated to another unconsumed contract, and all key assertions hold.

Conformance Only a restricted set of actions is allowed on a given contract.

Authorization The parties who may request a particular change are restricted.

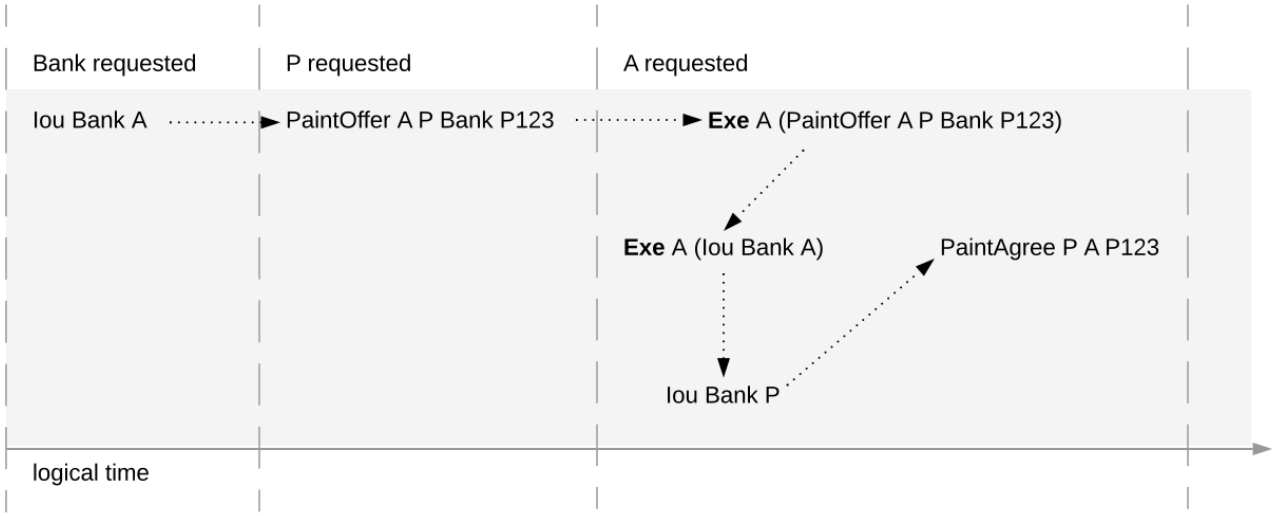
Only the last of these conditions depends on the party (or parties) requesting the change; the other two are general.

Consistency

Consistency consists of two parts:

1. *Contract consistency*: Contracts must be created before they are used, and they cannot be used once they are consumed.
2. *Key consistency*: Keys are unique and key assertions are satisfied.

To define this precisely, notions of before and after are needed. These are given by putting all actions in a sequence. Technically, the sequence is obtained by a pre-order traversal of the ledger's actions, noting that these actions form an (ordered) forest. Intuitively, it is obtained by always picking parent actions before their proper subactions, and otherwise always picking the actions on the left before the actions on the right. The image below depicts the resulting order on the paint offer example:



In the image, an action *act* happens before action *act'* if there is a (non-empty) path from *act* to *act'*. Then, *act'* happens after *act*.

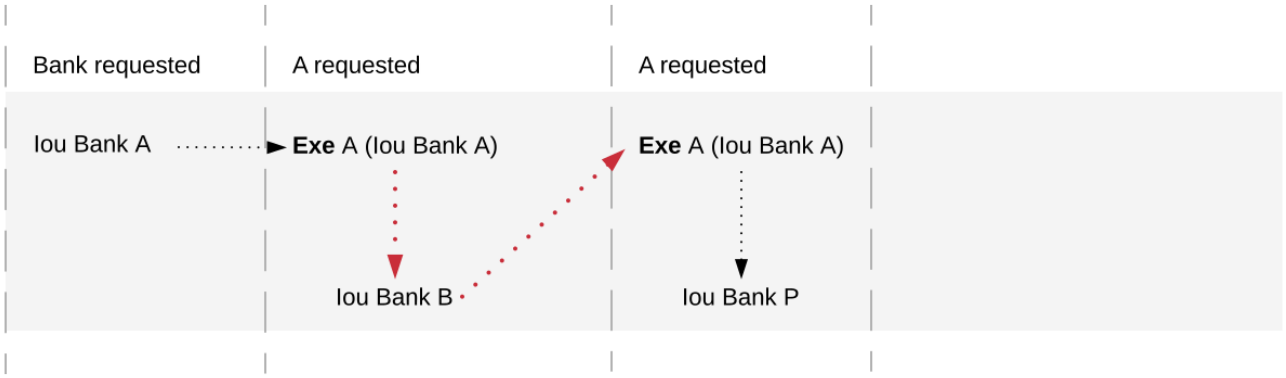
Contract Consistency

Contract consistency ensures that contracts are used after they have been created and before they are consumed.

Definition contract consistency A ledger is **consistent for a contract c** if all of the following holds for all actions *act* on *c*:

1. either *act* is itself **Create c** or a **Create c** happens before *act*
2. *act* does not happen before any **Create c** action
3. *act* does not happen after any **Exercise** action consuming *c*.

The consistency condition rules out the double spend example. As the red path below indicates, the second exercise in the example happens after a consuming exercise on the same contract, violating the contract consistency criteria.



In addition to the consistency notions, the before-after relation on actions can also be used to define the notion of **contract state** at any point in a given transaction. The contract state is changed by creating the contract and by exercising it consumingly. At any point in a transaction, we can then define the latest state change in the obvious way. Then, given a point in a transaction, the contract state of *c* is:

1. **active**, if the latest state change of *c* was a create;
2. **archived**, if the latest state change of *c* was a consuming exercise;
3. **inexistent**, if *c* never changed state.

A ledger is consistent for *c* exactly if **Exercise** and **Fetch** actions on *c* happen only when *c* is active, and **Create** actions only when *c* is inexistent. The figures below visualize the state of different contracts at all points in the example ledger.

The notion of order can be defined on all the different ledger structures: actions, transactions, lists of transactions, and ledgers. Thus, the notions of consistency, inputs and outputs, and contract state can also all be defined on all these structures. The **active contract set** of a ledger is the set of all contracts that are active on the ledger. For the example above, it consists of contracts *Iou Bank P* and *PaintAgree P A*.

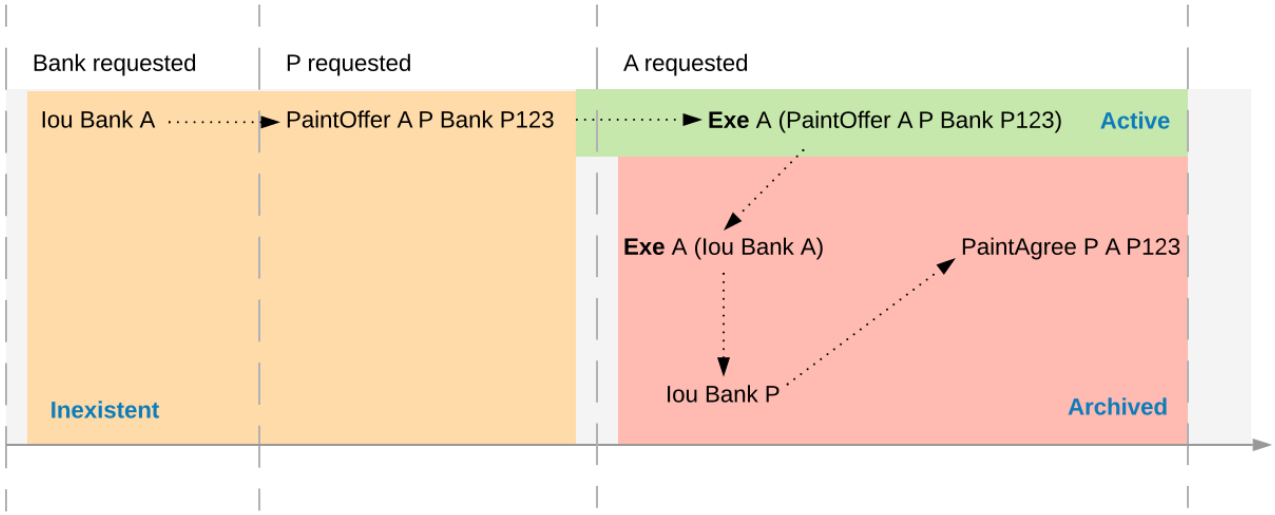


Fig. 25: Activeness of the *PaintOffer* contract

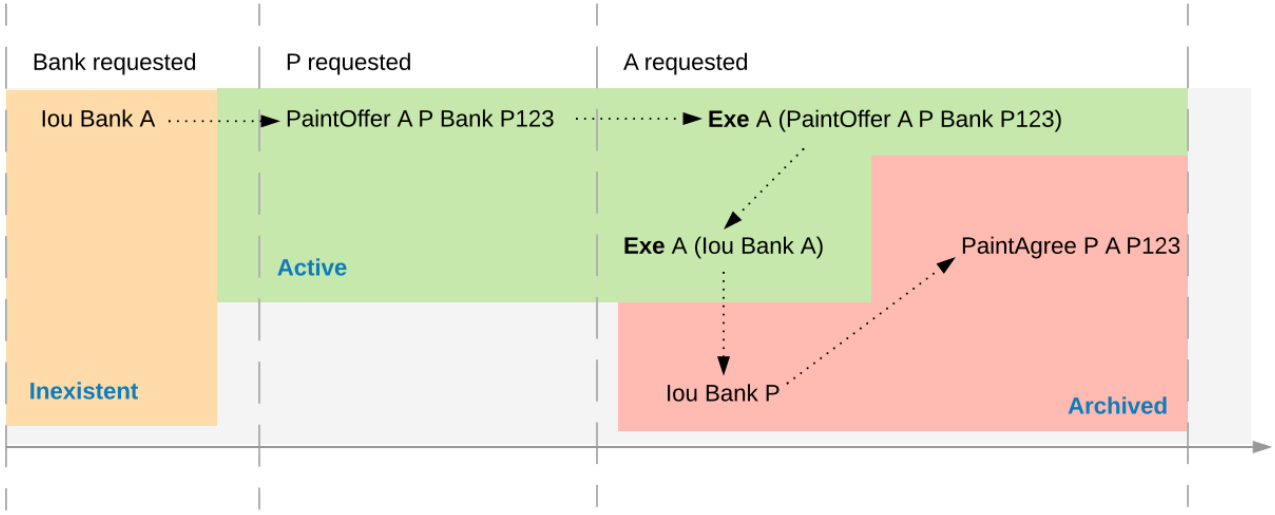


Fig. 26: Activeness of the *Iou Bank A* contract

Key Consistency

Contract keys introduce a key uniqueness constraint for the ledger. To capture this notion, the contract model must specify for every contract in the system whether the contract has a key and, if so, the key. Every contract can have at most one key.

Like contracts, every key has a state. An action *act* is an **action on a key** *k* if

> *act* is a **Create**, **Exercise**, or a **Fetch** action on a contract *c* with key *k*, or
> *act* is the key assertion **NoSuchKey** *k*.

Definition **key state** The **key state** of a key on a ledger is determined by the last action *act* on the key:

> If *act* is a **Create**, non-consuming **Exercise**, or **Fetch** action on a contract *c*, then the key state is **assigned** to *c*.
> If *act* is a consuming **Exercise** action or a **NoSuchKey** assertion, then the key state is **free**.
> If there is no such action *act*, then the key state is **unknown**.

A key is **unassigned** if its key state is either **free** or **unknown**.

Key consistency ensures that there is at most one active contract for each key and that all key assertions are satisfied.

Definition **key consistency** A ledger is **consistent for a key** *k* if for every action *act* on *k*, the key state *s* before *act* satisfies

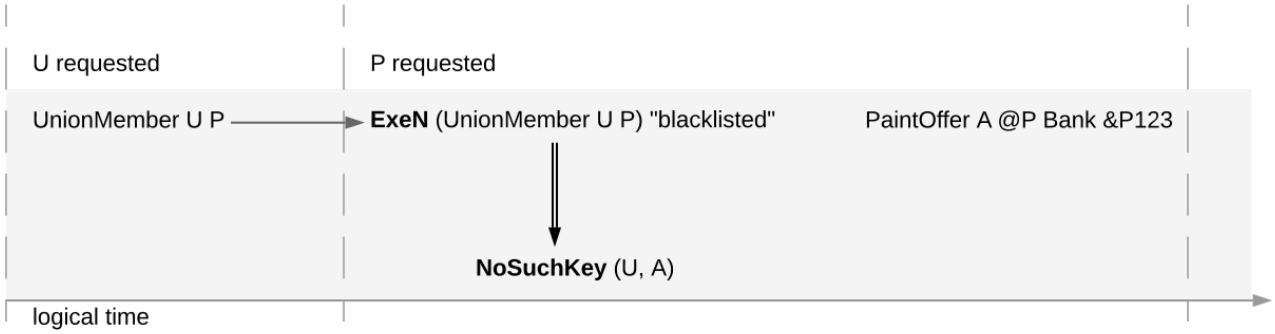
> If *act* is a **Create** action or **NoSuchKey** assertion, then *s* is **free** or **unknown**.
> If *act* is an **Exercise** or **Fetch** action on some contract *c*, then *s* is **assigned** to *c* or **unknown**.

Key consistency rules out the problematic examples around key consistency. For example, suppose that the painter *P* has made a paint offer to *A* with reference number *P123*, but *A* has not yet accepted it. When *P* tries to create another paint offer to *David* with the same reference number *P123*, then this creation action would violate key uniqueness. The following ledger violates key uniqueness for the key *(P, P123)*.

P requested	P requested
PaintOffer A @P Bank &P123	PaintOffer David @P Bank &P123

Key assertions can be used in workflows to evidence the inexistence of a certain kind of contract. For example, suppose that the painter *P* is a member of the union of painters *U*. This union maintains a blacklist of potential customers that its members must not do business with. A customer *A* is considered to be on the blacklist if there is an active contract *Blacklist @U &A*. To make sure that the painter *P* does not make a paint offer if *A* is blacklisted, the painter combines its commit with a **NoSuchKey** assertion on the key *(U, A)*. The following ledger shows the transaction, where *UnionMember U P* represents *P*'s membership in the union *U*. It grants *P* the choice to perform such an assertion, which is needed for *authorization*.

Key consistency extends to actions, transactions and lists of transactions just like the other consistency notions.

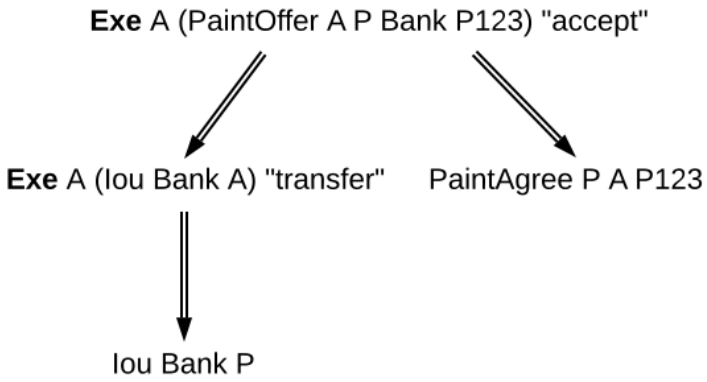


Ledger Consistency

Definition ledger consistency A ledger is **consistent** if it is consistent for all contracts and for all keys.

Internal Consistency

The above consistency requirement is too strong for actions and transactions in isolation. For example, the acceptance transaction from the paint offer example is not consistent as a ledger, because *PaintOffer A P Bank* and the *Iou Bank A* contracts are used without being created before:



However, the transaction can still be appended to a ledger that creates these contracts and yields a consistent ledger. Such transactions are said to be internally consistent, and contracts such as the *PaintOffer A P Bank P123* and *Iou Bank A* are called input contracts of the transaction. Dually, output contracts of a transaction are the contracts that a transaction creates and does not archive.

Definition internal consistency for a contract A transaction is **internally consistent for a contract c** if the following holds for all of its subactions *act* on the contract *c*

1. *act* does not happen before any **Create c** action
2. *act* does not happen after any exercise consuming *c*.

A transaction is **internally consistent** if it is internally consistent for all contracts and consistent for all keys.

Definition input contract For an internally consistent transaction, a contract *c* is an **input contract** of the transaction if the transaction contains an **Exercise** or a **Fetch** action on *c* but not a **Create c** action.

Definition **output contract** For an internally consistent transaction, a contract *c* is an **output contract** of the transaction if the transaction contains a **Create c** action, but not a consuming **Exercise** action on *c*.

Note that the input and output contracts are undefined for transactions that are not internally consistent. The image below shows some examples of internally consistent and inconsistent transactions.

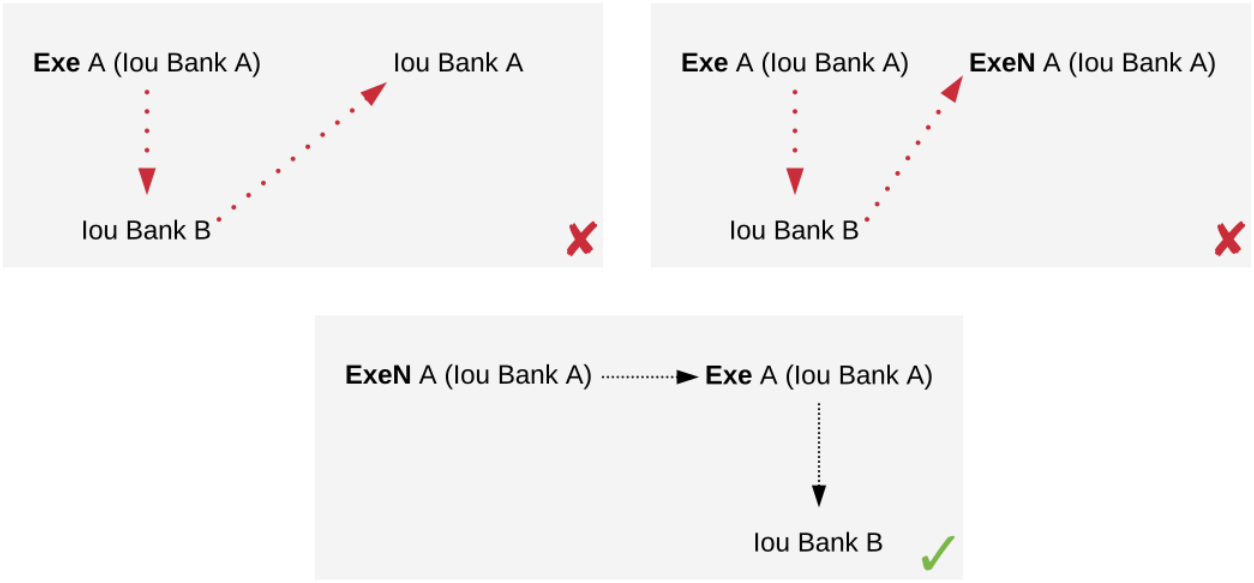


Fig. 27: The first two transactions violate the conditions of internal consistency. The first transaction creates the *Iou* after exercising it consumingly, violating both conditions. The second transaction contains a (non-consuming) exercise on the *Iou* after a consuming one, violating the second condition. The last transaction is internally consistent.

Similar to input contracts, we define the input keys as the set that must be unassigned at the beginning of a transaction.

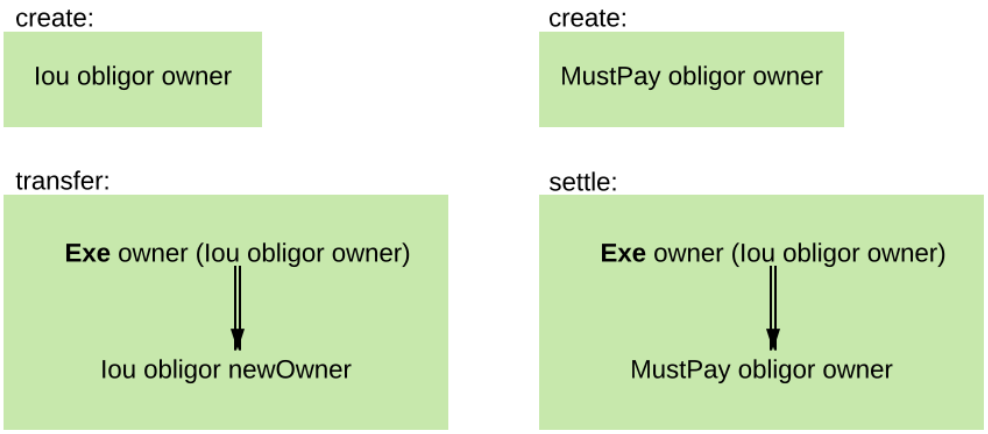
Definition **input key** A key *k* is an **input key** to an internally consistent transaction if the first action *act* on *k* is either a **Create** action or a **NoSuchKey** assertion.

In the *blacklisting example*, *P*'s transaction has two input keys: *(U, A)* due to the **NoSuchKey** action and *(P, P123)* as it creates a *PaintOffer* contract.

Conformance

The *conformance* condition constrains the actions that may occur on the ledger. This is done by considering a **contract model** *M* (or a **model** for short), which specifies the set of all possible actions. A ledger is **conformant to M** (or conforms to M) if all top-level actions on the ledger are members of *M*. Like consistency, the notion of conformance does not depend on the requesters of a commit, so it can also be applied to transactions and lists of transactions.

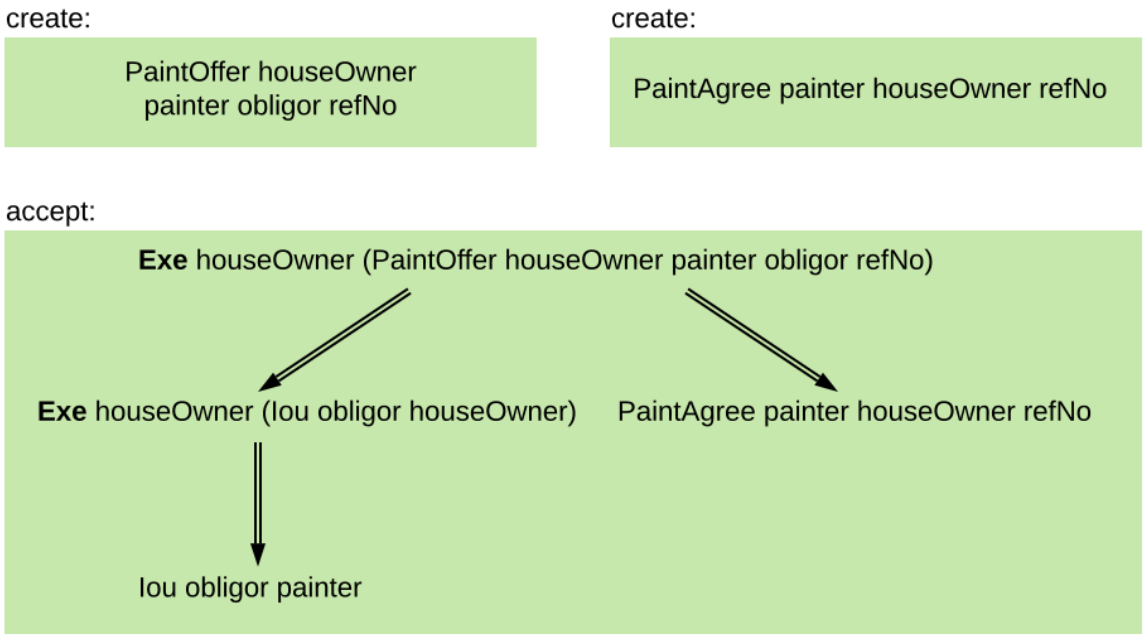
For example, the set of allowed actions on IOU contracts could be described as follows.



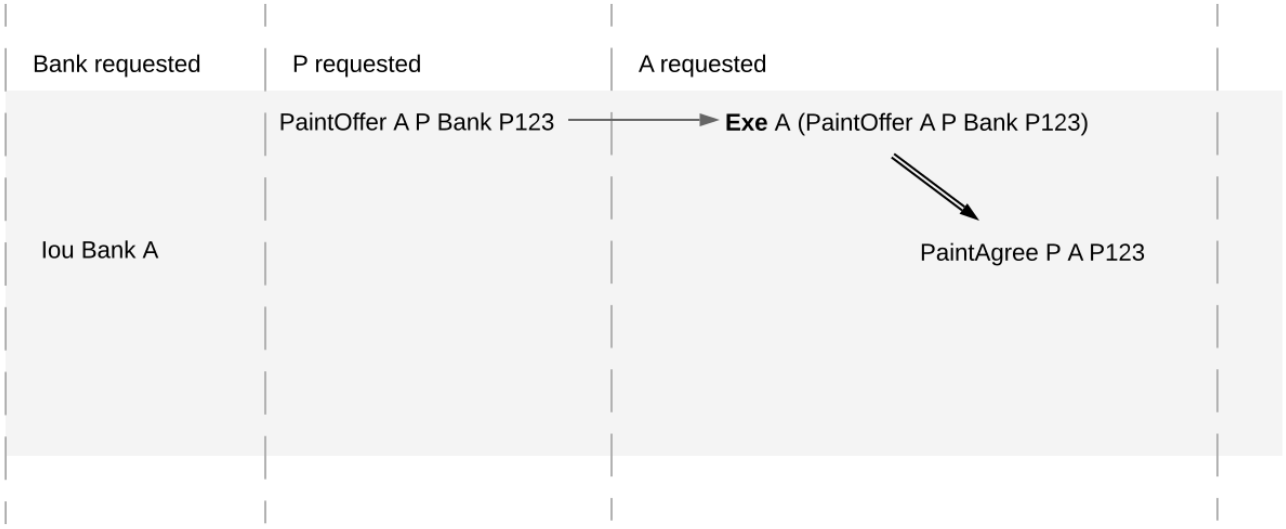
The boxes in the image are templates in the sense that the contract parameters in a box (such as obligor or owner) can be instantiated by arbitrary values of the appropriate type. To facilitate understanding, each box includes a label describing the intuitive purpose of the corresponding set of actions. As the image suggests, the transfer box imposes the constraint that the bank must remain the same both in the exercised IOU contract, and in the newly created IOU contract. However, the owner can change arbitrarily. In contrast, in the settle actions, both the bank and the owner must remain the same. Furthermore, to be conformant, the actor of a transfer action must be the same as the owner of the contract.

Of course, the constraints on the relationship between the parameters can be arbitrarily complex, and cannot conveniently be reproduced in this graphical representation. This is the role of Daml – it provides a much more convenient way of representing contract models. The link between Daml and contract models is explained in more detail in a *later section*.

To see the conformance criterion in action, assume that the contract model allows only the following actions on *PaintOffer* and *PaintAgree* contracts.



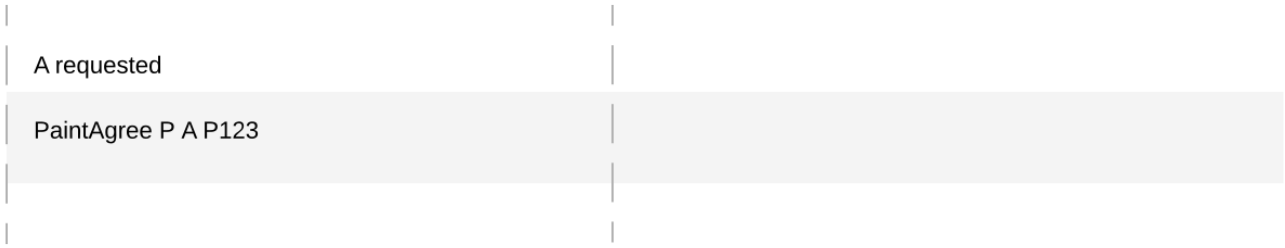
The problem with the example where Alice changes the offer's outcome to avoid transferring the money now becomes apparent.



A's commit is not conformant to the contract model, as the model does not contain the top-level action she is trying to commit.

Authorization

The last criterion rules out the last two problematic examples, *an obligation imposed on a painter*, and *the painter stealing Alice's money*. The first of those is visualized below.



The reason why the example is intuitively impermissible is that the *PaintAgree* contract is supposed to express that the painter has an obligation to paint Alice's house, but he never agreed to that obligation. On paper contracts, obligations are expressed in the body of the contract, and imposed on the contract's *signatories*.

Signatories, Agreements, and Maintainers

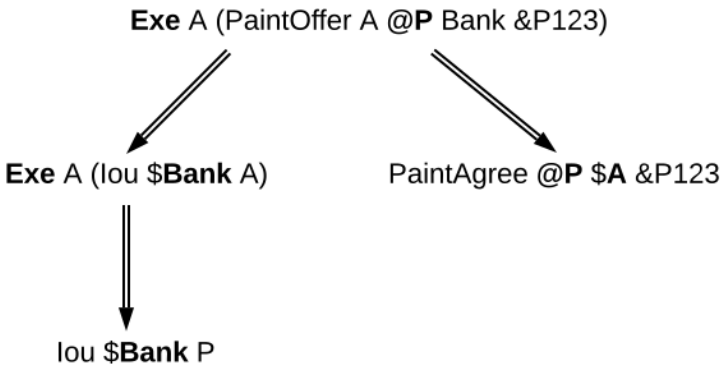
To capture these elements of real-world contracts, the **contract model** additionally specifies, for each contract in the system:

1. A non-empty set of **signatories**, the parties bound by the contract.
2. An optional **agreement text** associated with the contract, specifying the off-ledger, real-world obligations of the signatories.
3. If the contract is associated with a key, a non-empty set of **maintainers**, the parties that make sure that at most one unconsumed contract exists for the key. The maintainers must be a subset of the signatories and depend only on the key. This dependence is captured by the function *maintainers* that takes a key and returns the key's maintainers.

In the example, the contract model specifies that

1. an *Iou obligor owner* contract has only the *obligor* as a signatory, and no agreement text.
2. a *MustPay obligor owner* contract has both the *obligor* and the *owner* as signatories, with an agreement text requiring the obligor to pay the owner a certain amount, off the ledger.
3. a *PaintOffer houseOwner painter obligor refNo* contract has only the painter as the signatory, with no agreement text. Its associated key consists of the painter and the reference number. The painter is the maintainer.
4. a *PaintAgree houseOwner painter refNo* contract has both the house owner and the painter as signatories, with an agreement text requiring the painter to paint the house. The key consists of the painter and the reference number. The painter is the only maintainer.

In the graphical representation below, signatories of a contract are indicated with a dollar sign (as a mnemonic for an obligation) and use a bold font. Maintainers are marked with @ (as a mnemonic who enforces uniqueness). Since maintainers are always signatories, parties marked with @ are implicitly signatories. For example, annotating the paint offer acceptance action with signatories yields the image below.



Authorization Rules

Signatories allow one to precisely state that the painter has an obligation. The imposed obligation is intuitively invalid because the painter did not agree to this obligation. In other words, the painter did not *authorize* the creation of the obligation.

In a Daml ledger, a party can **authorize** a subaction of a commit in either of the following ways:

> Every top-level action of the commit is authorized by all requesters of the commit.
> Every consequence of an exercise action *act* on a contract *c* is authorized by all signatories of *c* and all actors of *act*.

The second authorization rule encodes the offer-acceptance pattern, which is a prerequisite for contract formation in contract law. The contract *c* is effectively an offer by its signatories who act as offerers. The exercise is an acceptance of the offer by the actors who are the offerees. The consequences of the exercise can be interpreted as the contract body so the authorization rules of Daml ledgers closely model the rules for contract formation in contract law.

A commit is **well-authorized** if every subaction *act* of the commit is authorized by at least all of the **required authorizers** of *act*, where:

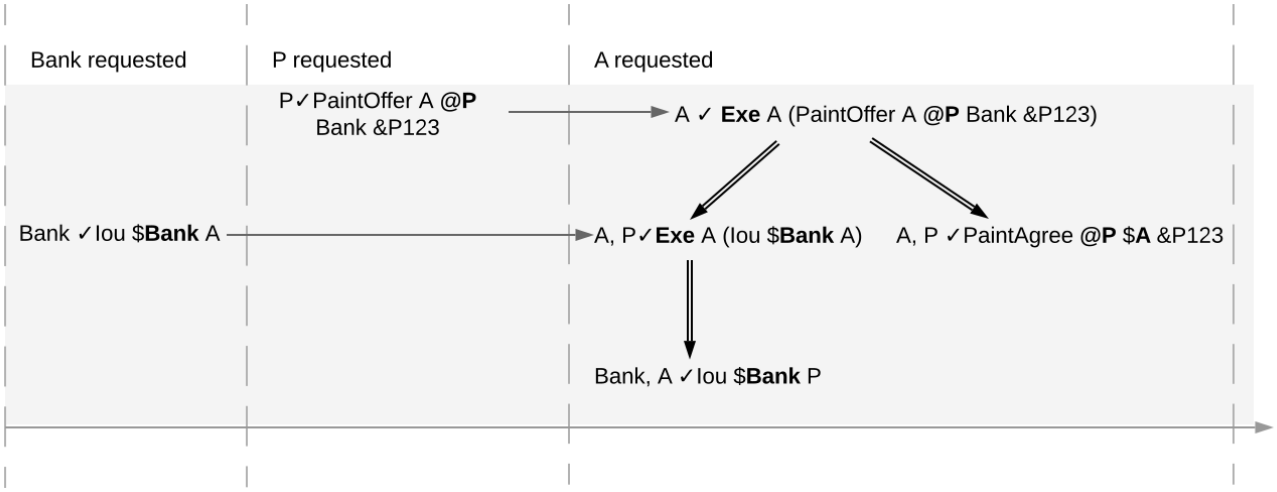
1. the required authorizers of a **Create** action on a contract *c* are the signatories of *c*.
2. the required authorizers of an **Exercise** or a **Fetch** action are its actors.
3. the required authorizers of a **NoSuchKey** assertion are the maintainers of the key.

We lift this notion to ledgers, whereby a ledger is well-authorized exactly when all of its commits are.

Examples

An intuition for how the authorization definitions work is most easily developed by looking at some examples. The main example, the paint offer ledger, is intuitively legitimate. It should therefore also be well-authorized according to our definitions, which it is indeed.

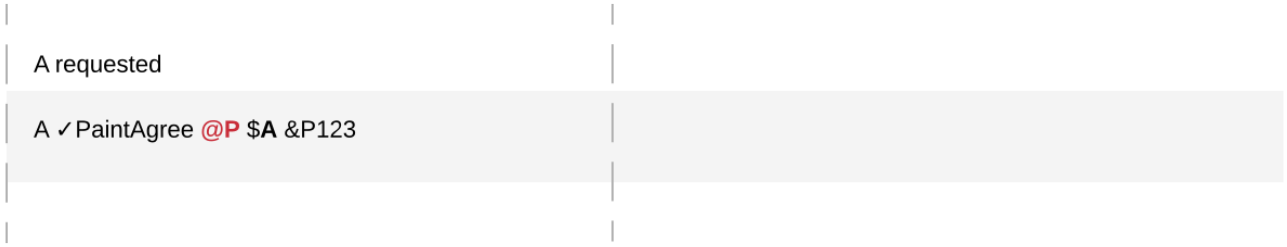
In the visualizations below, ✓ *act* denotes that the parties authorize the action *act*. The resulting authorizations are shown below.



In the first commit, the bank authorizes the creation of the IOU by requesting that commit. As the bank is the sole signatory on the IOU contract, this commit is well-authorized. Similarly, in the second commit, the painter authorizes the creation of the paint offer contract, and painter is the only signatory on that contract, making this commit also well-authorized.

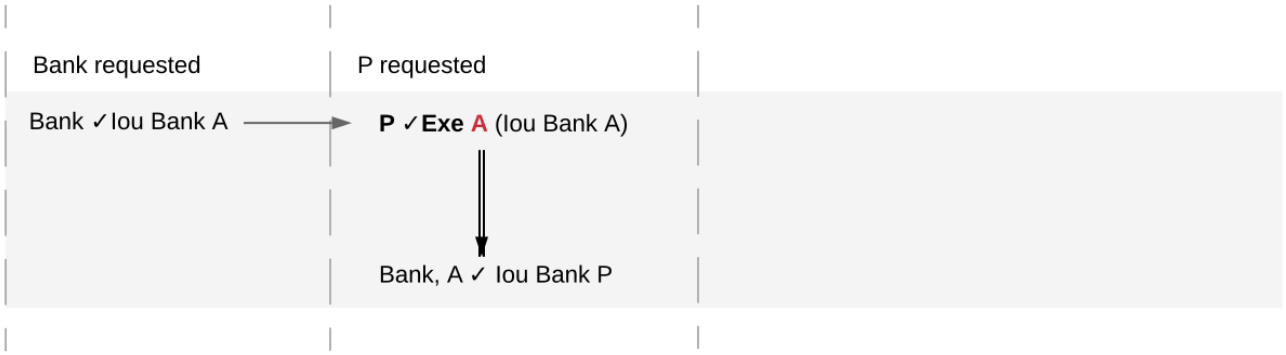
The third commit is more complicated. First, Alice authorizes the exercise on the paint offer by requesting it. She is the only actor on this exercise, so this complies with the authorization requirement. Since the painter is the signatory of the paint offer, and Alice the actor of the exercise, they jointly authorize all consequences of the exercise. The first consequence is an exercise on the IOU, with Alice as the actor, so this is permissible. The second consequence is the creation of the new IOU (for P) by exercising the old IOU (for A). As the IOU was formerly signed by the bank, with Alice as the actor of the exercise, they jointly authorize this creation. This action is permissible as the bank is the sole signatory. The final consequence is creating the paint agreement with Alice and the painter as signatories. Since they both authorize the action, this is also permissible. Thus, the entire third commit is also well-authorized, and so is the ledger.

Similarly, the intuitively problematic examples are prohibited by our authorization criterion. In the first example, Alice forced the painter to paint her house. The authorizations for the example are shown below.



Alice authorizes the **Create** action on the *PaintAgree* contract by requesting it. However, the painter is also a signatory on the *PaintAgree* contract, but he did not authorize the **Create** action. Thus, this ledger is indeed not well-authorized.

In the second example, the painter steals money from Alice.



The bank authorizes the creation of the IOU by requesting this action. Similarly, the painter authorizes the exercise that transfers the IOU to him. However, the actor of this exercise is Alice, who has not authorized the exercise. Thus, this ledger is not well-authorized.

The rationale for making the maintainers required authorizers for a **NoSuchKey** assertion is discussed in the next section about *privacy*.

Valid Ledgers, Obligations, Offers and Rights

Daml ledgers are designed to mimic real-world interactions between parties, which are governed by contract law. The validity conditions on the ledgers, and the information contained in contract models have several subtle links to the concepts of the contract law that are worth pointing out.

First, in addition to the explicit off-ledger obligations specified in the agreement text, contracts also specify implicit **on-ledger obligations**, which result from consequences of the exercises on contracts. For example, the *PaintOffer* contains an on-ledger obligation for *A* to transfer her IOU in case she accepts the offer. Agreement texts are therefore only necessary to specify obligations that are not already modeled as permissible actions on the ledger. For example, *P*'s obligation to paint the house cannot be sensibly modeled on the ledger, and must thus be specified by the agreement text.

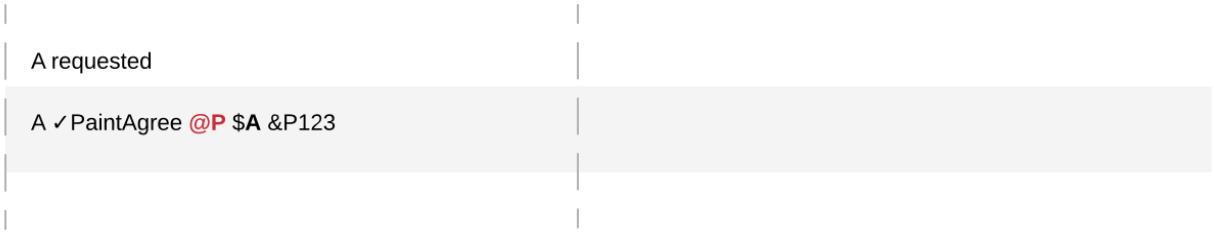
Second, every contract on a Daml ledger can simultaneously model both:

a real-world offer, whose consequences (both on- and off-ledger) are specified by the **Exercise** actions on the contract allowed by the contract model, and
a real-world contract proper , specified through the contract's (optional) agreement text.

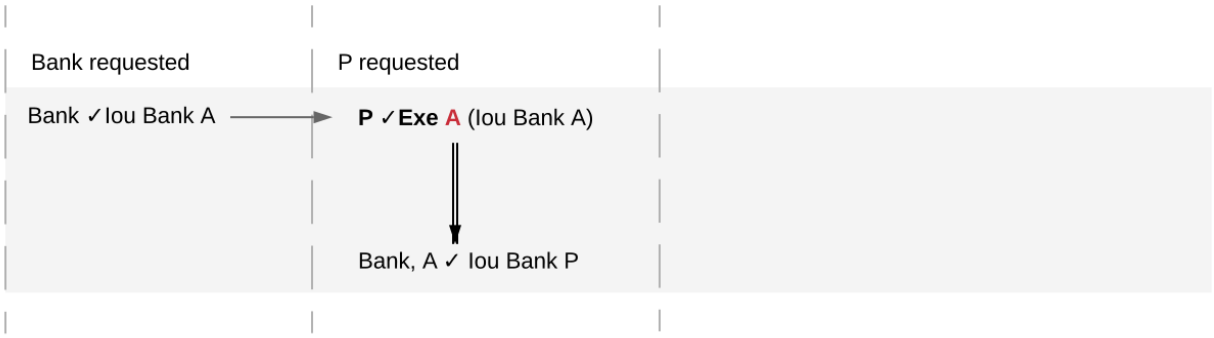
Third, in Daml ledgers, as in the real world, one person's rights are another person's obligations. For example, *A*'s right to accept the *PaintOffer* is *P*'s obligation to paint her house in case she accepts. In Daml ledgers, a party's rights according to a contract model are the exercise actions the party can perform according to the authorization and conformance rules.

Finally, validity conditions ensure three important properties of the Daml ledger model, that mimic the contract law.

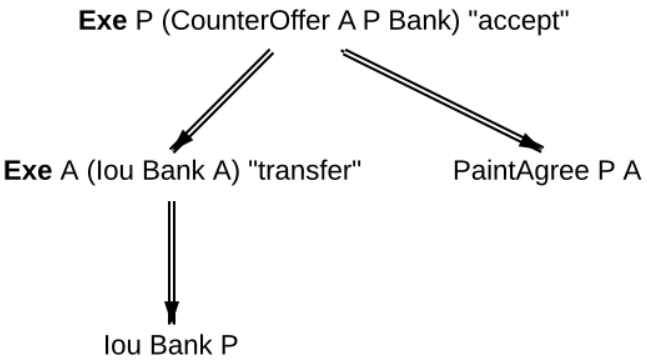
1. **Obligations need consent**. Daml ledgers follow the offer-acceptance pattern of the contract law, and thus ensures that all ledger contracts are formed voluntarily. For example, the following ledger is not valid.



2. **Consent is needed to take away on-ledger rights**. As only **Exercise** actions consume contracts, the rights cannot be taken away from the actors; the contract model specifies exactly who the actors are, and the authorization rules require them to approve the contract consumption. In the examples, Alice had the right to transfer her IOUs; painter's attempt to take that right away from her, by performing a transfer himself, was not valid.



Parties can still **delegate** their rights to other parties. For example, assume that Alice, instead of accepting painter's offer, decides to make him a counteroffer instead. The painter can then accept this counteroffer, with the consequences as before:



Here, by creating the *CounterOffer* contract, Alice delegates her right to transfer the IOU contract to the painter. In case of delegation, prior to submission, the requester must get informed about the contracts that are part of the requested transaction, but where the requester is not a signatory. In the example above, the painter must learn about the existence of the IOU for Alice before he can request the acceptance of the *CounterOffer*. The concepts of observers and divulgence, introduced in the next section, enable such scenarios.

3. **On-ledger obligations cannot be unilaterally escaped**. Once an obligation is recorded on a Daml ledger, it can only be removed in accordance with the contract model. For example, assuming the IOU contract model shown earlier, if the ledger records the creation of a *MustPay* contract, the bank cannot later simply record an action that consumes this contract:

That is, this ledger is invalid, as the action above is not conformant to the contract model.

1.44.1.3 Causality and Local Daml Ledgers

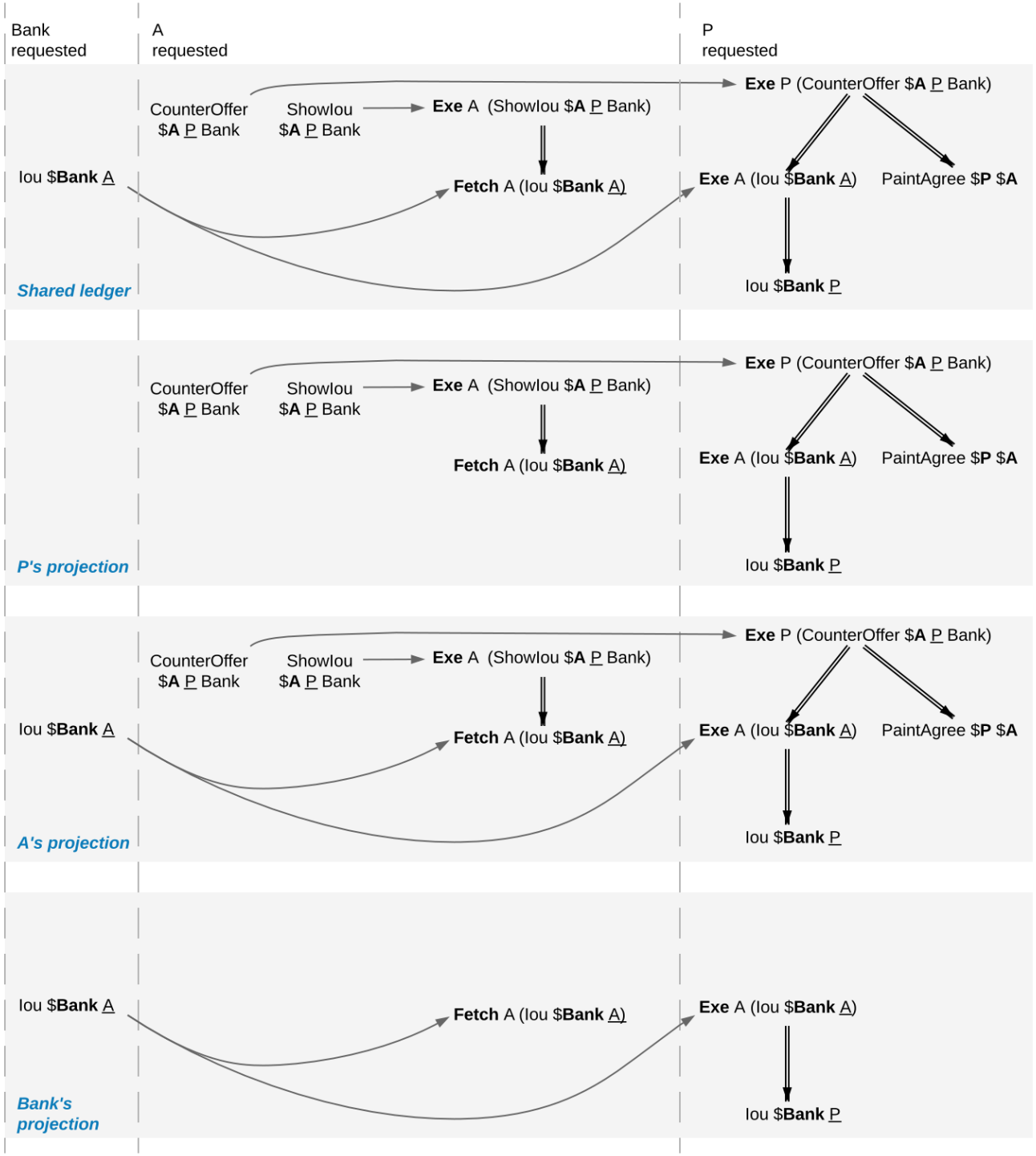
Daml ledgers do not totally order all transactions. So different parties may observe two transactions on different Participant Nodes in different orders via the *Ledger API*. Moreover, different Participant Nodes may output two transactions for the same party in different orders. This document explains the ordering guarantees that Daml ledgers do provide, by *example* and formally via the concept of *causality graphs* and *local ledgers*.

The presentation assumes that you are familiar with the following concepts:

The *Ledger API*
The *Daml Ledger Model*

Causality Examples

A Daml Ledger need not totally order all transaction, unlike ledgers in the Daml Ledger Model. The following examples illustrate these ordering guarantees of the Ledger API. They are based on the paint counteroffer workflow from the Daml Ledger Model's *privacy section*, ignoring the total ordering coming from the Daml Ledger Model. Recall that *the party projections* are as follows.


Bank
requested
A
requested
P
requested
CounterOffer
$A P Bank
ShowIou
$A P Bank
Exe A (ShowIou $A P Bank)
Exe P (CounterOffer $A P Bank)
Iou $Bank A
Fetch A (Iou $Bank A)
Exe A (Iou $Bank A)
PaintAgree $P $A
Iou $Bank P
Shared ledger
P's projection
A's projection
Bank's
projection

Stakeholders of a Contract See Creation and Archival in the Same Order

Every Daml Ledger orders the creation of the *CounterOffer A P Bank* before the painter exercising the consuming choice on the *CounterOffer*. (If the **Create** was ordered after the **Exercise**, the resulting shared ledger would be inconsistent, which violates the validity guarantee of Daml ledgers.) Accordingly, Alice will see the creation before the archival on her transaction stream and so will the painter. This does not depend on whether they are hosted on the same Participant Node.

Signatories of a Contract and Stakeholder Actors See Usages After the Creation and Before the Archival

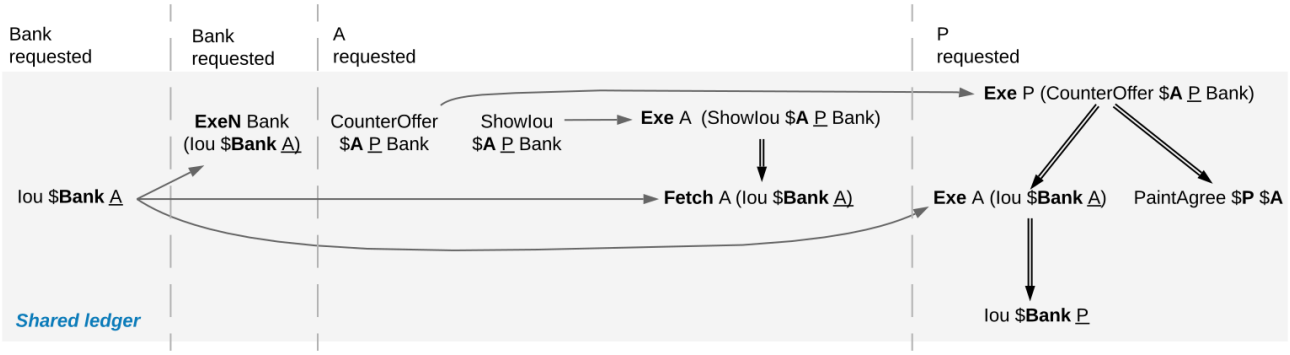
The *Fetch A (Iou Bank A)* action comes after the creation of the *Iou Bank A* and before its archival, for both Alice and the Bank, because the Bank is a signatory of the *Iou Bank A* contract and Alice is a stakeholder of the *Iou Bank A* contract and an actor on the **Fetch** action.

Commits Are Atomic

Alice sees the **Create** of her *Iou* before the creation of the *CounterOffer*, because the *CounterOffer* is created in the same commit as the **Fetch** of the *Iou* and the **Fetch** commit comes after the **Create** of the *Iou*.

Non-Consuming Usages in Different Commits May Appear in Different Orders

Suppose that the Bank exercises a non-consuming choice on the *Iou Bank A* without consequences while Alice creates the *CounterOffer*. In the ledger shown below, the Bank's commit comes before Alice's commit.



The Bank's projection contains the nonconsuming **Exercise** and the **Fetch** action on the *Iou*. Yet, the **Fetch** may come before the non-consuming **Exercise** in the Bank's transaction tree stream.

Out-of-Band Causality Is Not Respected

The following examples assume that Alice splits up her commit into two as follows:

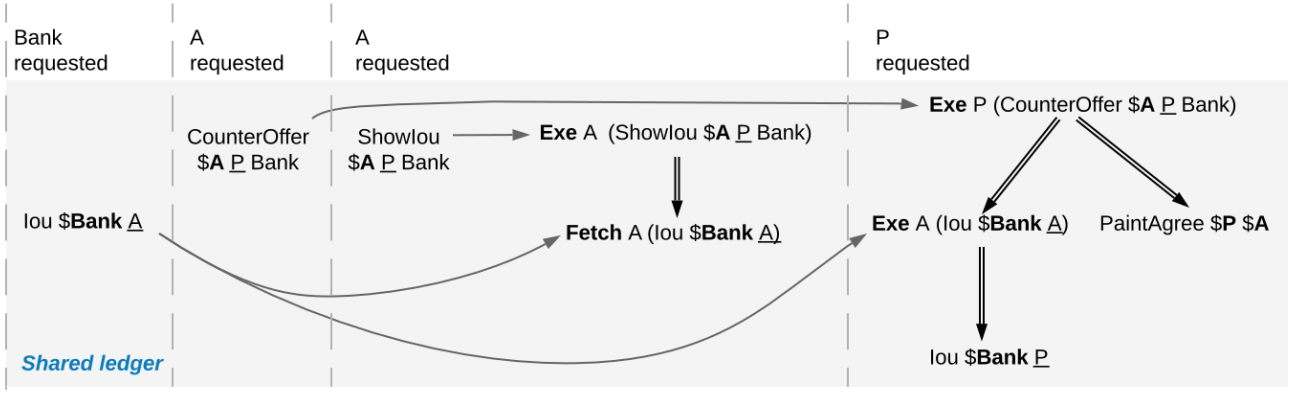


Fig. 28: Counteroffer workflow with four commits.

Alice can specify in the *CounterOffer* the *Iou* that she wants to pay the painter with. In a deployed implementation, Alice's application first observes the created *Iou* contract via the transaction service or active contract service before she requests to create the *CounterOffer*. Such application logic does not induce an ordering between commits. So the creation of the *CounterOffer* need not come after the creation of the *Iou*.

If Alice is hosted on several Participant Nodes, the Participant Nodes can therefore output the two creations in either order.

The rationale for this behaviour is that Alice could have learnt about the contract ID out of band or made it up. The Participant Nodes therefore cannot know whether there will ever be a **Create** event for the contract. So if Participant Nodes delayed outputting the **Create** action for the *CounterOffer* until a **Create** event for the *Iou* contract was published, this delay might last forever and liveness is lost. Daml ledgers therefore do not capture data flow through applications.

Divulged Actions Do Not Induce Order

The painter witnesses the fetching of Alice's *Iou* when the *ShowIou* contract is consumed. The painter also witnesses the **Exercise** of the *Iou* when Alice exercises the transfer choice as a consequence of the painter accepting the *CounterOffer*. However, as the painter is not a stakeholder of Alice's *Iou* contract, he may observe Alice's *ShowIou* commit after the archival of the *Iou* as part of accepting the *CounterOffer*.

In practice, this can happen in a setup where two Participant Nodes N_1 and N_2 host the painter. He sees the divulged *Iou* and the created *CounterOffer* through N_1's transaction tree stream and then submits the acceptance through N_1. Like in the previous example, N_2 does not know about the dependence of the two commits. Accordingly, N_2 may output the accepting transaction *before* the *ShowIou* contract on the transaction stream.

Even though this may seem unexpected, it is in line with stakeholder-based ledgers: Since the painter is not a stakeholder of the *Iou* contract, he should not care about the archivals or creates of the contract. In fact, the divulged *Iou* contract shows up neither in the painter's active contract service nor in the flat transaction stream. Such witnessed events are included in the transaction tree

stream as a convenience: They relieve the painter from computing the consequences of the choice and enable him to check that the action conforms to the Daml model.

Similarly, being an actor of an **Exercise** action induces order with respect to other uses of the contract only if the actor is a contract stakeholder. This is because non-stakeholder actors of an **Exercise** action merely authorize the action, but they do not track whether the contract is active; this is what signatories and contract observers are for. Analogously, choice observers of an **Exercise** action benefit from the ordering guarantees only if they are contract stakeholders.

The Ordering Guarantees Depend on the Party

By the previous example, for the painter, fetching the *Iou* is not ordered before transferring the *Iou*. For Alice, however, the **Fetch** must appear before the **Exercise** because Alice is a stakeholder on the *Iou* contract. This shows that the ordering guarantees depend on the party.

Causality Graphs

The above examples indicate that Daml ledgers order transactions only partially. Daml ledgers can be represented as finite directed acyclic graphs (DAG) of transactions.

Definition causality graph A **causality graph** is a finite directed acyclic graph G of transactions that is transitively closed. Transitively closed means that whenever v_1 -> v_2 and v_2 -> v_3 are edges in G, then there is also an edge v_1 -> v_3 in G.

Definition action order For a causality graph G, the induced **action order** on the actions in the transactions combines the graph-induced order between transactions with the execution order of actions inside each transaction. It is the least partial order that includes the following ordering relations between two actions act_1 and act_2:

act_1 and act_2 belong to the same transaction and act_1 precedes act_2 in the transaction.
act_1 and act_2 belong to different transactions in vertices tx_1 and tx_2 and there is a path in G from tx_1 to tx_2.

Note: Checking for an *edge* instead of a *path* in G from tx_1 to tx_2 is equivalent because causality graphs are transitively closed. The definition uses *path* because the figures below omit transitive edges for readability.

The action order is a partial order on the actions in a causality graph. For example, the following diagram shows such a causality graph for the ledger in the above *Out-of-band causality example*. Each grey box represents one transaction and the graph edges are the solid arrows between the boxes. Diagrams omit transitive edges for readability; in this graph the edge from *tx1* to *tx4* is not shown. The **Create** action of Alice's *Iou* is ordered before the **Create** action of the *ShowIou* contract because there is an edge from the transaction *tx1* with the *Iou* **Create** to the transaction *tx3* with the *ShowIou* **Create**. Moreover, the *ShowIou* **Create** action is ordered before the **Fetch** of Alice's *Iou* because the **Create** action precedes the **Fetch** action in the transaction. In contrast, the **Create** actions of the *CounterOffer* and Alice's *Iou* are unordered: neither precedes the other because they belong to different transaction and there is no directed path between them.

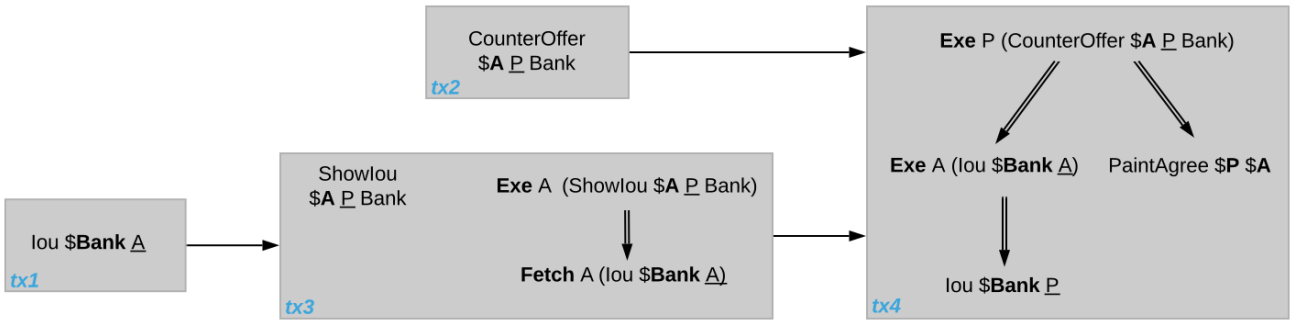


Fig. 29: Causality graph for the *counteroffer workflow with four commits*.

Consistency

Consistency ensures that a causality graph sufficiently orders all the transactions. It generalizes *ledger consistency* from the Daml Ledger Model as *explained below*.

Definition **Causal consistency for a contract** Let *G* be a causality graph and *X* be a set of actions on a contract *c* that belong to transactions in *G*. The graph *G* is **causally consistent for the contract** *c* on *X* if all of the following hold:

If *X* is not empty, then *X* contains exactly one **Create** action. This action precedes all other actions in *X* in *G*'s action order.

If *X* contains a consuming **Exercise** action *act*, then *act* follows all actions in *X* other than *act* in *G*'s action order.

Definition **Causal consistency for a key** Let *G* be a causality graph and *X* be a set of actions on a key *k* that belong to transactions in *G*. The graph *G* is **causally consistent for the key** *k* on *X* if all of the following hold:

All **Create** and consuming **Exercise** actions in *X* are totally ordered in *G*'s action order and **Create**s and consuming **Exercise**s alternate, starting with **Create**. Every consecutive **Create**-**Exercise** pair acts on the same contract.

All **NoSuchKey** actions in *X* are action-ordered with respect to all **Create** and consuming **Exercise** actions in *X*. No **NoSuchKey** action is action-ordered between a **Create** action and its subsequent consuming **Exercise** action in *X*.

Definition **Consistency for a causality graph** Let *X* be a subset of the actions in a causality graph *G*. Then *G* is **consistent** on *X* (or *X*-**consistent**) if *G* is causally consistent for all contracts *c* on the set of actions on *c* in *X* and for all keys *k* on the set of actions on *k* in *X*. *G* is **consistent** if *G* is consistent on all the actions in *G*.

When edges are added to an *X*-consistent causality graph such that it remains acyclic and transitively closed, the resulting graph is again *X*-consistent. So it makes sense to consider minimal consistent causality graphs.

Definition **Minimal consistent causality graph** An *X*-consistent causality graph *G* is *X*-**minimal** if no strict subgraph of *G* (same vertices, fewer edges) is an *X*-consistent causality graph. If *X* is the set of all actions in *G*, then *X* is omitted.

For example, the *above causality graph for the split counteroffer workflow* is consistent. This causality graph is minimal, as the following analysis shows:

Edge	Justification
tx1 -> *tx3*	Alice's *Iou* **Create** action of must precede the **Fetch** action.
tx2 -> *tx4*	The *CounterOffer* **Create** action of must precede the **Exercise** action.
tx3 -> *tx4*	The consuming **Exercise** action on Alice's *Iou* must follow the **Fetch** action.

We can focus on parts of the causality graph by restricting the set *X*. If *X* consists of the actions on *Iou* contracts, this causality graph is *X*-consistent. Yet, it is not *X*-minimal since the edge *tx2* -> *tx4* can be removed without violating *X*-consistency: the edge is required only because of the *CounterOffer* actions, which are excluded from *X*. The *X*-minimal consistent causality graph looks as follows, where the actions in *X* are highlighted in red.

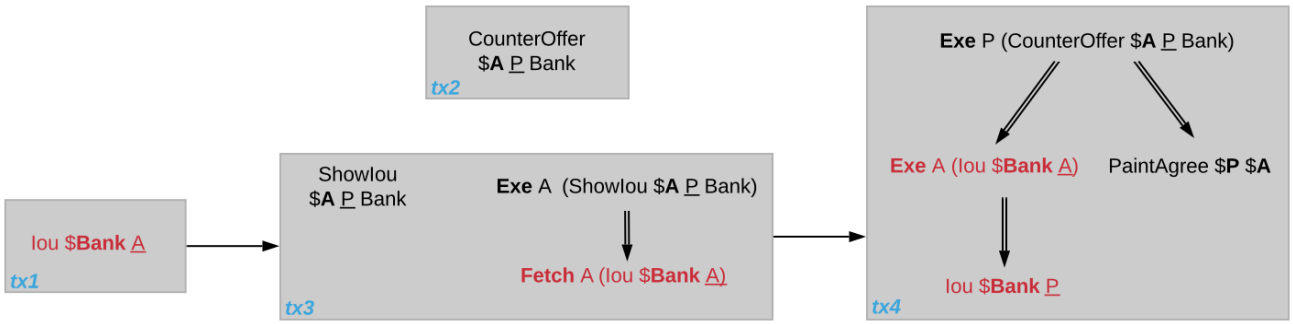
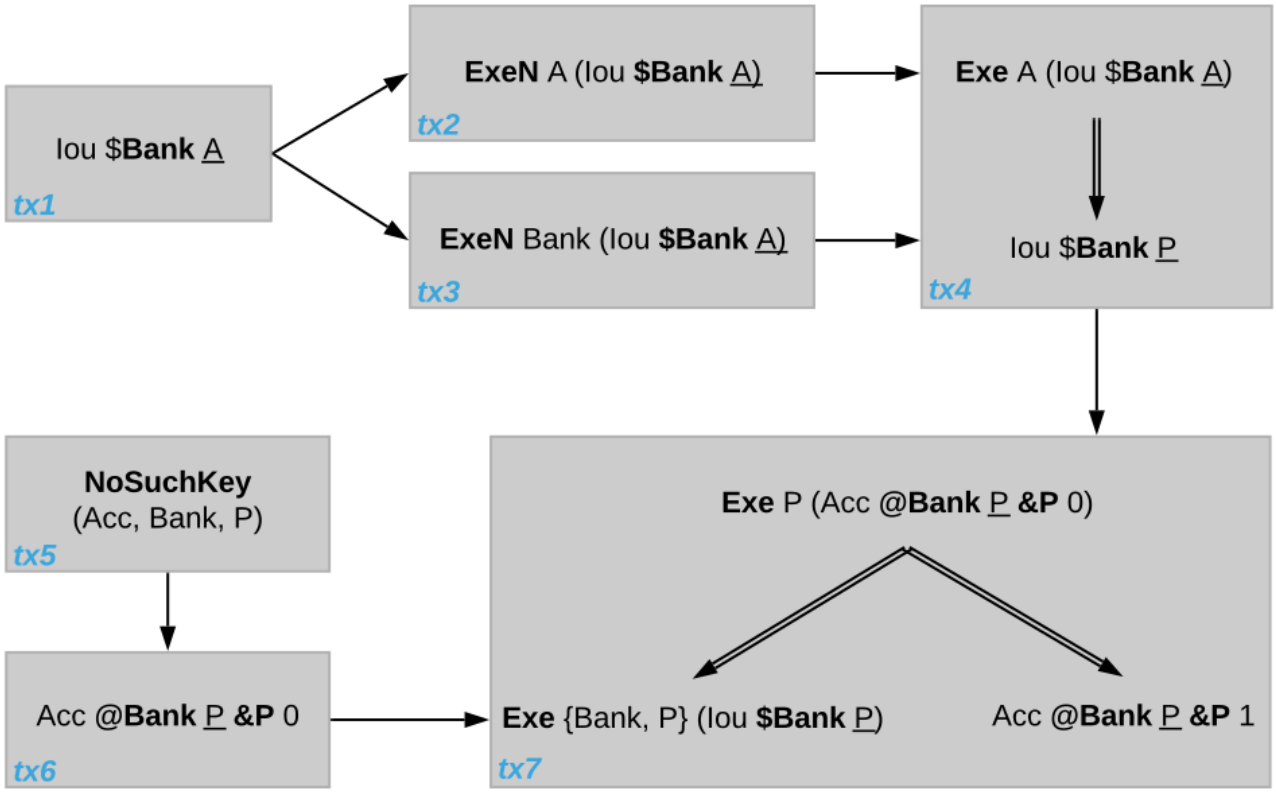


Fig. 30: Minimal consistent causality graph for the highlighted actions.

Another example of a minimal causality graph is shown below. At the top, the transactions *tx1* to *tx4* create an *Iou* for Alice, exercise two non-consuming choices on it, and transfer the *Iou* to the painter. At the bottom, *tx5* asserts that there is no key for an Account contract for the painter. Then, *tx6* creates an such account with balance 0 and *tx7* deposits the painter's *Iou* from *tx4* into the account, updating the balance to 1.



Unlike in a linearly ordered ledger, the causality graph relates the transactions of the *Iou* transfer workflow with the *Account* creation workflow only at the end, when the *Iou* is deposited into the account. As will be formalized below, the Bank, Alice, and the painter therefore need not observe the transactions *tx1* to *tx7* in the same order.

Moreover, transaction *tx2* and *tx3* are unordered in this causality graph even though they act on the same *Iou* contract. However, as both actions are non-consuming, they do not interfere with each other and could therefore be parallelized, too. Alice and the Bank accordingly may observe them in different orders.

The **NoSuchKey** action in *tx5* must be ordered with respect to the two Account **Create** actions in *tx6* and *tx7* and the consuming **Exercise** on the Account contract in *tx7*, by the key consistency conditions. For this set of transactions, consistency allows only one such order: *tx5* comes before *tx6* because *tx7* is atomic: *tx5* cannot be interleaved with *tx7*, e.g., between the consuming **Exercise** of the *Acc Bank P P 0* and the **Create** of the updated account *Acc Bank P P 1*.

NoSuchKey actions are similar to non-consuming **Exercise**s and **Fetch**es of contracts when it comes to causal ordering: If there were another transaction *tx5'* with a **NoSuchKey** *(Acc, Bank, P)* action, then *tx5* and *tx5'* need not be ordered, just like *tx2* and *tx3* are unordered.

From Causality Graphs to Ledgers

Since causality graphs are acyclic, their vertices can be sorted topologically and the resulting list is again a causality graph, where every vertex has an outgoing edge to all later vertices. If the original causality graph is *X*-consistent, then so is the topological sort, as topological sorting merely adds edges. For example, the transactions on the *ledger* in the *out-of-band causality example* are a topological sort of the *corresponding causality graph*.

Conversely, we can reduce an *X*-consistent causality graph to only the causal dependencies that *X*-consistency imposes. This gives a minimal *X*-consistent causality graph.

Definition Reduction of a consistent causality graph For an *X*-consistent causality graph *G*, there exists a unique minimal *X*-consistent causality graph $reduce_X(G)$ with the same vertices and the edges being a subset of *G*. The graph $reduce_X(G)$ is called the *X*-**reduction** of *G*. As before, *X* is omitted if it contains all actions in *G*.

The causality graph for the split *CounterOffer* workflow is minimal and therefore its own reduction. It is also the reduction of the topological sort, i.e., the *ledger* in the *out-of-band causality example*.

Note: The reduction $reduce_X(G)$ of an *X*-consistent causality graph *G* can be computed as follows:

1. Set the vertices of *G'* to the vertices of *G*.
2. The causal consistency conditions for contracts and keys demand that certain pairs of actions act_1 and act_2 in *X* must be action-ordered. For each such pair, determine the actions' ordering in *G* and add an edge to *G'* from the earlier action's transaction to the later action's transaction.
3. $reduce_X(G)$ is the transitive closure of *G'*.

Topological sort and reduction link causality graphs *G* to the ledgers *L* from the Daml Ledger Model. Topological sort transforms a causality graph *G* into a sequence of transactions; extending them with the requesters gives a sequence of commits, i.e., a ledger in the Daml Ledger Model. Conversely, a sequence of commits *L* yields a causality graph G_L by taking the transactions as vertices and adding an edge from *tx1* to *tx2* whenever *tx1*'s commit precedes *tx2*'s commit in the sequence.

There are now two consistency definitions:

> *Ledger Consistency* according to Daml Ledger Model
> *Consistency of causality graph*

Fortunately, the two definitions are equivalent: If *G* is a consistent causality graph, then the topological sort is ledger consistent. Conversely, if the sequence of commits *L* is ledger consistent, G_L is a consistent causality graph, and so is the reduction $reduce(G_L)$.

Local Ledgers

As explained in the Daml Ledger Model, parties see only a *projection* of the shared ledger for privacy reasons. Like consistency, projection extends to causality graphs as follows.

Definition Stakeholder informee A party *P* is a **stakeholder informee** of an action *act* if all of the following holds:

> *P* is an informee of *act*.
> If *act* is an action on a contract then *P* is a stakeholder of the contract.

An **Exercise** and **Fetch** action acts on the input contract, a **Create** action on the created contract, and a **NoSuchKey** action does not act on a contract. So for a **NoSuchKey** action, the stakeholder informees are the key maintainers.

Definition Causal consistency for a party A causality graph *G* is **consistent for a party** *P* (*P*-consistent) if *G* is consistent on all the actions that *P* is a stakeholder informee of.

The notions of *X*-minimality and *X*-reduction extend to parties accordingly.

For example, the *split counteroffer causality graph without the edge tx2 -> tx4* is consistent for the Bank because the Bank is a stakeholder informee of exactly the highlighted actions. It is also minimal Bank-consistent and the Bank-reduction of the *original split counteroffer causality graph*.

Definition Projection of a consistent causality graph The **projection** $proj_P(G)$ of a consistent causality graph *G* to a party *P* is the *P*-reduction of the following causality graph *G'*:

The vertices of *G'* are the vertices of *G* projected to *P*, excluding empty projections.

There is an edge between two vertices v_1 and v_2 in *G'* if there is an edge from the *G*-vertex corresponding to v_1 to the *G*-vertex corresponding to v_2.

For the *split counteroffer causality graph*, the projections to Alice, the Bank, and the painter are as follows.

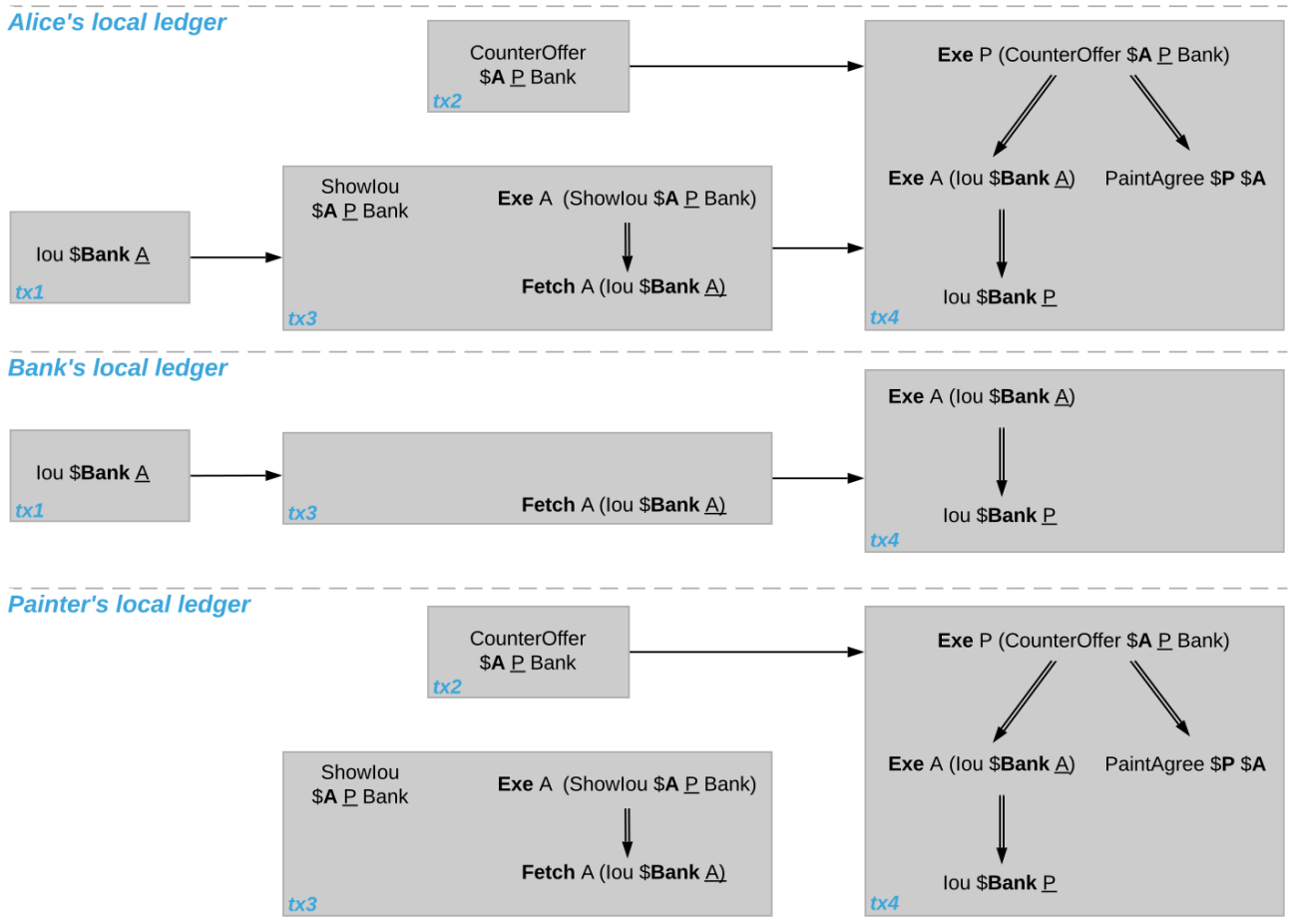


Fig. 31: Projections of the *split counteroffer causality graph*.

Alice's projection is the same as the original minimal causality graph. The Bank sees only actions on *Iou* contracts, so the causality graph projection does not contain *tx2* any more. Similarly, the painter

is not aware of *tx1*, where Alice's *Iou* is created. Moreover, there is no longer an edge from *tx3* to *tx4* in the painter's local ledger. This is because the edge is induced by the **Fetch** of Alice's *Iou* preceding the consuming **Exercise**. However, the painter is not an informee of those two actions; he merely witnesses the **Fetch** and **Exercise** actions as part of divulgence. Therefore no ordering is required from the painter's point of view. This difference explains the *divulgence causality example*.

Ledger API Ordering Guarantees

The *Transaction Service* provides the updates as a stream of Daml transactions and the *Active Contract Service* summarizes all the updates up to a given point by the contracts that are active at this point. Conceptually, both services are derived from the local ledger that the Participant Node manages for each hosted party. That is, the transaction tree stream for a party is a topological sort of the party's local ledger. The flat transaction stream contains precisely the `CreatedEvent`s and `ArchivedEvent`s that correspond to **Create** and consuming **Exercise** actions in transaction trees on the transaction tree stream where the party is a stakeholder of the affected contract.

Note: The transaction trees of the *Transaction Service* omit **Fetch** and **NoSuchKey** actions that are part of the transactions in the local ledger. The **Fetch** and **NoSuchKey** actions are thus removed before the *Transaction Service* outputs the transaction trees.

Similarly, the active contract service provides the set of contracts that are active at the returned offset according to the Transaction Service streams. That is, the contract state changes of all events from the transaction event stream are taken into account in the provided set of contracts. In particular, an application can process all subsequent events from the flat transaction stream or the transaction tree stream without having to take events before the snapshot into account.

Since the topological sort of a local ledger is not unique, different Participant Nodes may pick different orders for the transaction streams of the same party. Similarly, the transaction streams for different parties may order common transactions differently, as the party's local ledgers impose different ordering constraints. Nevertheless, Daml ledgers ensure that all local ledgers are projections of a virtual shared causality graph that connects to the Daml Ledger Model as described above. The ledger validity guarantees therefore extend via the local ledgers to the Ledger API. These guarantees are subject to the deployed Daml ledger's trust assumptions.

Note: The virtual shared causality graph exists only as a concept, to reason about Daml ledger guarantees. A deployed Daml ledger in general does not store or even construct such a shared causality graph. The Participant Nodes merely maintain the local ledgers for their parties. They synchronize these local ledgers to the extent that they remain consistent. That is, all the local ledgers can in theory be combined into a consistent single causality graph of which they are projections.

Explaining the Causality Examples

The *causality examples* can be explained in terms of causality graphs and local ledgers as follows:

1. *Stakeholders of a Contract See Creation and Archival in the Same Order* Causal consistency for the contract requires that the **Create** comes before the consuming **Exercise** action on the contract. As all stakeholders are informees on **Create** and consuming **Exercise** actions of their contracts, the stakeholder's local ledgers impose this order on the actions.
2. *Signatories of a Contract and Stakeholder Actors See Usages After the Creation and Before the Archival* Causal consistency for the contract requires that the **Create** comes before the non-consuming **Exercise** and **Fetch** actions of a contract and that consuming **Exercise**s follow them. Since signatories and stakeholder actors are informees of **Create**, **Exercise**, and **Fetch** actions, the stakeholder's local ledgers impose this order on the actions.
3. *Commits Are Atomic* Local ledgers are DAGs of (projected) transactions. Topologically sorting such a DAG cannot interleave one transaction with another, even if the transaction consists of several top-level actions.
4. *Non-Consuming Usages in Different Commits May Appear in Different Orders* Causal consistency does not require ordering between non-consuming usages of a contract. As there is no other action in the transaction that would prescribe an ordering, the Participant Nodes can output them in any order.
5. *Out-of-Band Causality Is Not Respected* Out-of-band data flow is not captured by causal consistency and therefore does not induce ordering.
6. *Divulged Actions Do Not Induce Order* The painter is not an informee of the **Fetch** and **Exercise** actions on Alice's *Iou*; he merely witnesses them. The *painter's local ledger* therefore does not order *tx3* before *tx4*. So the painter's transaction stream can output *tx4* before *tx3*.
7. *The Ordering Guarantees Depend on the Party* Alice is an informee of the **Fetch** and **Exercise** actions on her *Iou*. Unlike for the painter, *her local ledger* does order *tx3* before *tx4*, so Alice is guaranteed to observe *tx3* before *tx4* on all Participant Nodes through which she is connect to the Daml ledger.

1.44.1.4 Privacy

The previous sections have addressed two out of three questions posed in the introduction: what the ledger looks like, and who may request which changes. This section addresses the last one, who sees which changes and data. That is, it explains the privacy model for Daml ledgers.

The privacy model of Daml Ledgers is based on a **need-to-know basis**, and provides privacy **on the level of subtransactions**. Namely, a party learns only those parts of ledger changes that affect contracts in which the party has a stake, and the consequences of those changes. And maintainers see all changes to the contract keys they maintain.

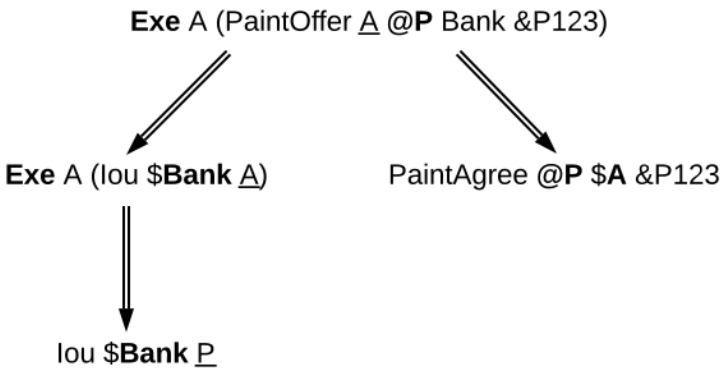
To make this more precise, a stakeholder concept is needed.

Contract Observers and Stakeholders

Intuitively, as signatories are bound by a contract, they have a stake in it. Actors might not be bound by the contract, but they still have a stake in their actions, as these are the actor's rights. Generalizing this, **observers** are parties who might not be bound by the contract, but still have the right to see the contract. For example, Alice should be an observer of the *PaintOffer*, such that she is made aware that the offer exists.

Signatories are already determined by the contract model discussed so far. The full **contract model** additionally specifies the **contract observers** on each contract. A **stakeholder** of a contract (according to a given contract model) is then either a signatory or a contract observer on the contract. Note that in Daml, as detailed *later*, controllers specified using simple syntax are automatically made contract observers whenever possible.

In the graphical representation of the paint offer acceptance below, contract observers who are not signatories are indicated by an underline.



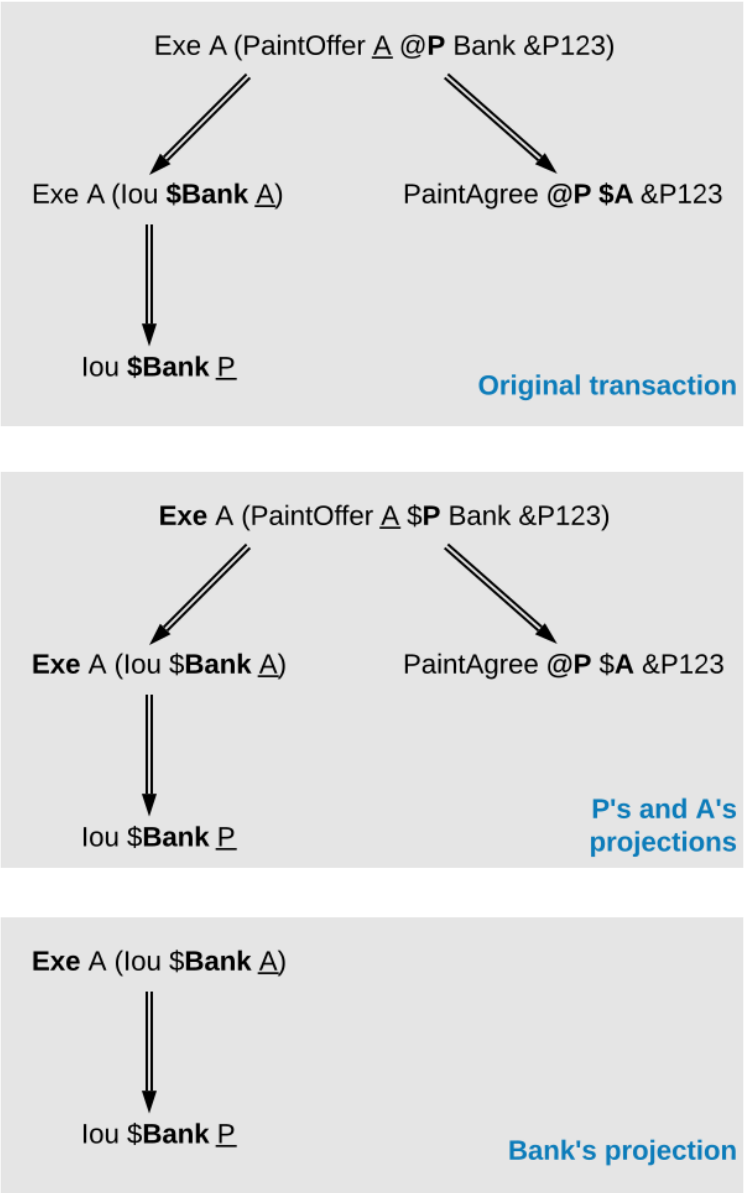
Choice Observers

In addition to contract observers, the contract model can also specify **choice observers** on individual **Exercise** actions. Choice observers get to see a specific exercise on a contract, and to view its consequences. Choice observers are not considered stakeholders of the contract, they only affect the set of informees on an action, for the purposes of projection (see below).

Projections

Stakeholders should see changes to contracts they hold a stake in, but that does not mean that they have to see the entirety of any transaction that their contract is involved in. This is made precise through *projections* of a transaction, which define the view that each party gets on a transaction. Intuitively, given a transaction within a commit, a party will see only the subtransaction consisting of all actions on contracts where the party is a stakeholder. Thus, privacy is obtained on the subtransaction level.

An example is given below. The transaction that consists only of Alice's acceptance of the *PaintOffer* is projected for each of the three parties in the example: the painter, Alice, and the bank.



Since both the painter and Alice are stakeholders of the *PaintOffer* contract, the exercise on this contract is kept in the projection of both parties. Recall that consequences of an exercise action are a part of the action. Thus, both parties also see the exercise on the *Iou Bank A* contract, and the creations of the *Iou Bank P* and *PaintAgree* contracts.

The bank is *not* a stakeholder on the *PaintOffer* contract (even though it is mentioned in the contract). Thus, the projection for the bank is obtained by projecting the consequences of the exercise on the *PaintOffer*. The bank is a stakeholder in the contract *Iou Bank A*, so the exercise on this contract is kept in the bank's projection. Lastly, as the bank is not a stakeholder of the *PaintAgree* contract, the corresponding **Create** action is dropped from the bank's projection.

Note the privacy implications of the bank's projection. While the bank learns that a transfer has occurred from *A* to *P*, the bank does *not* learn anything about *why* the transfer occurred. In practice, this means that the bank does not learn what *A* is paying for, providing privacy to *A* and *P* with respect to the bank.

As a design choice, Daml Ledgers show to contract observers only the *state changing* actions on the contract. More precisely, **Fetch** and non-consuming **Exercise** actions are not shown to contract ob-

servers - except when they are also actors or choice observers of these actions. This motivates the following definition: a party *p* is an **informee** of an action *A* if one of the following holds:

A is a **Create** on a contract *c* and *p* is a stakeholder of *c*.
A is a consuming **Exercise** on a contract *c*, and *p* is a stakeholder of *c* or an actor on *A*. Note that a Daml flexible controller *can be an exercise actor without being a contract stakeholder*.
A is a non-consuming **Exercise** on a contract *c*, and *p* is a signatory of *c* or an actor on *A*.
A is an **Exercise** action and *p* is a choice observer on *A*.
A is a **Fetch** on a contract *c*, and *p* is a signatory of *c* or an actor on *A*.
A is a **NoSuchKey** *k* assertion and *p* is a maintainer of *k*.

Then, we can formally define the **projection** of a transaction $tx = act_1, \ldots, act_n$ for a party *p* is the sub-transaction obtained by doing the following for each action act_i:

1. If *p* is an informee of act_i, keep act_i as-is.
2. Else, if act_i has consequences, replace act_i by the projection (for *p*) of its consequences, which might be empty.
3. Else, drop act_i.

Finally, the **projection of a ledger** *l* for a party *p* is a list of transactions obtained by first projecting the transaction of each commit in *l* for *p*, and then removing all empty transactions from the result. Note that the projection of a ledger is not a ledger, but a list of transactions. Projecting the ledger of our complete paint offer example yields the following projections for each party:

Examine each party's projection in turn:

1. The painter does not see any part of the first commit, as he is not a stakeholder of the *Iou Bank A* contract. Thus, this transaction is not present in the projection for the painter at all. However, the painter is a stakeholder in the *PaintOffer*, so he sees both the creation and the exercise of this contract (again, recall that all consequences of an exercise action are a part of the action

itself).
2. Alice is a stakeholder in both the *Iou Bank A* and *PaintOffer A B Bank* contracts. As all top-level actions in the ledger are performed on one of these two contracts, Alice's projection includes all the transactions from the ledger intact.
3. The Bank is only a stakeholder of the IOU contracts. Thus, the bank sees the first commit's transaction as-is. The second commit's transaction is, however dropped from the bank's projection. The projection of the last commit's transaction is as described above.

Ledger projections do not always satisfy the definition of consistency, even if the ledger does. For example, in P's view, *Iou Bank A* is exercised without ever being created, and thus without being made active. Furthermore, projections can in general be non-conformant. However, the projection for a party *p* is always

> internally consistent for all contracts,
> consistent for all contracts on which *p* is a stakeholder, and
> consistent for the keys that *p* is a maintainer of.

In other words, *p* is never a stakeholder on any input contracts of its projection. Furthermore, if the contract model is **subaction-closed**, which means that for every action *act* in the model, all subactions of *act* are also in the model, then the projection is guaranteed to be conformant. As we will see shortly, Daml-based contract models are conformant. Lastly, as projections carry no information about the requesters, we cannot talk about authorization on the level of projections.

Privacy Through Authorization

Setting the maintainers as required authorizers for a **NoSuchKey** assertion ensures that parties cannot learn about the existence of a contract without having a right to know about their existence. So we use authorization to impose *access controls* that ensure confidentiality about the existence of contracts. For example, suppose now that for a *PaintAgreement* contract, both signatories are key maintainers, not only the painter. That is, we consider *PaintAgreement @A @P &P123* instead of *PaintAgreement $A @P &P123*. Then, when the painter's competitor *Q* passes by *A*'s house and sees that the house desperately needs painting, *Q* would like to know whether there is any point in spending marketing efforts and making a paint offer to *A*. Without key authorization, *Q* could test whether a ledger implementation accepts the action **NoSuchKey** *(A, P, refNo)* for different guesses of the reference number *refNo*. In particular, if the ledger does not accept the transaction for some *refNo*, then *Q* knows that *P* has some business with *A* and his chances of *A* accepting his offer are lower. Key authorization prevents this flow of information because the ledger always rejects *Q*'s action for violating the authorization rules.

For these access controls, it suffices if one maintainer authorizes a **NoSuchKey** assertion. However, we demand that *all* maintainers must authorize it. This is to prevent spam in the projection of the maintainers. If only one maintainer sufficed to authorize a key assertion, then a valid ledger could contain **NoSuchKey** *k* assertions where the maintainers of *k* include, apart from the requester, arbitrary other parties. Unlike **Create** actions to contract observers, such assertions are of no value to the other parties. Since processing such assertions may be expensive, they can be considered spam. Requiring all maintainers to authorize a **NoSuchKey** assertion avoids the problem.

Divulgence: When Non-Stakeholders See Contracts

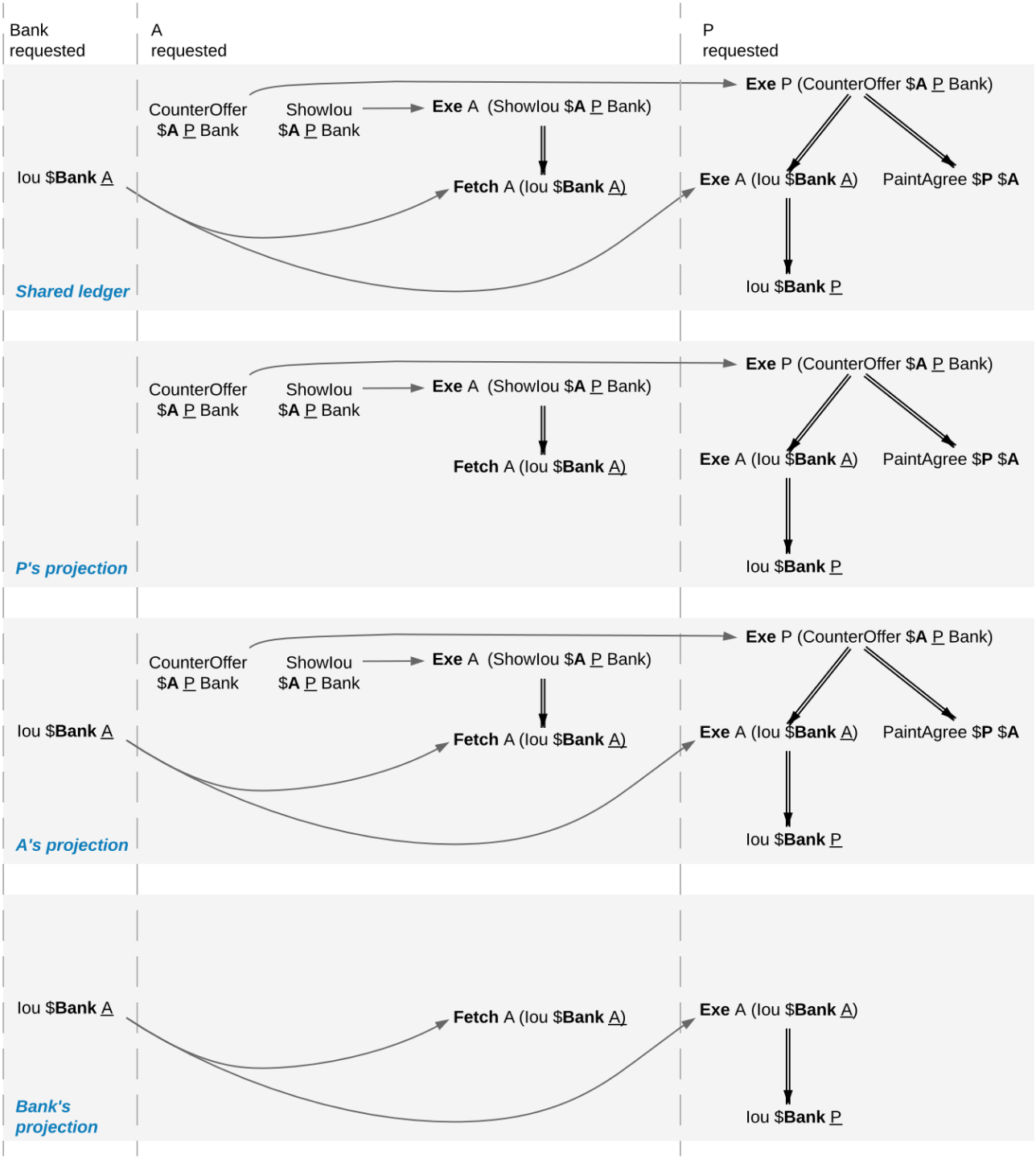
The guiding principle for the privacy model of Daml ledgers is that contracts should only be shown to their stakeholders. However, ledger projections can cause contracts to become visible to other parties as well.

In the example of *ledger projections of the paint offer*, the exercise on the *PaintOffer* is visible to both the painter and Alice. As a consequence, the exercise on the *Iou Bank A* is visible to the painter, and the creation of *Iou Bank P* is visible to Alice. As actions also contain the contracts they act on, *Iou Bank A* was thus shown to the painter and *Iou Bank P* was shown to Alice.

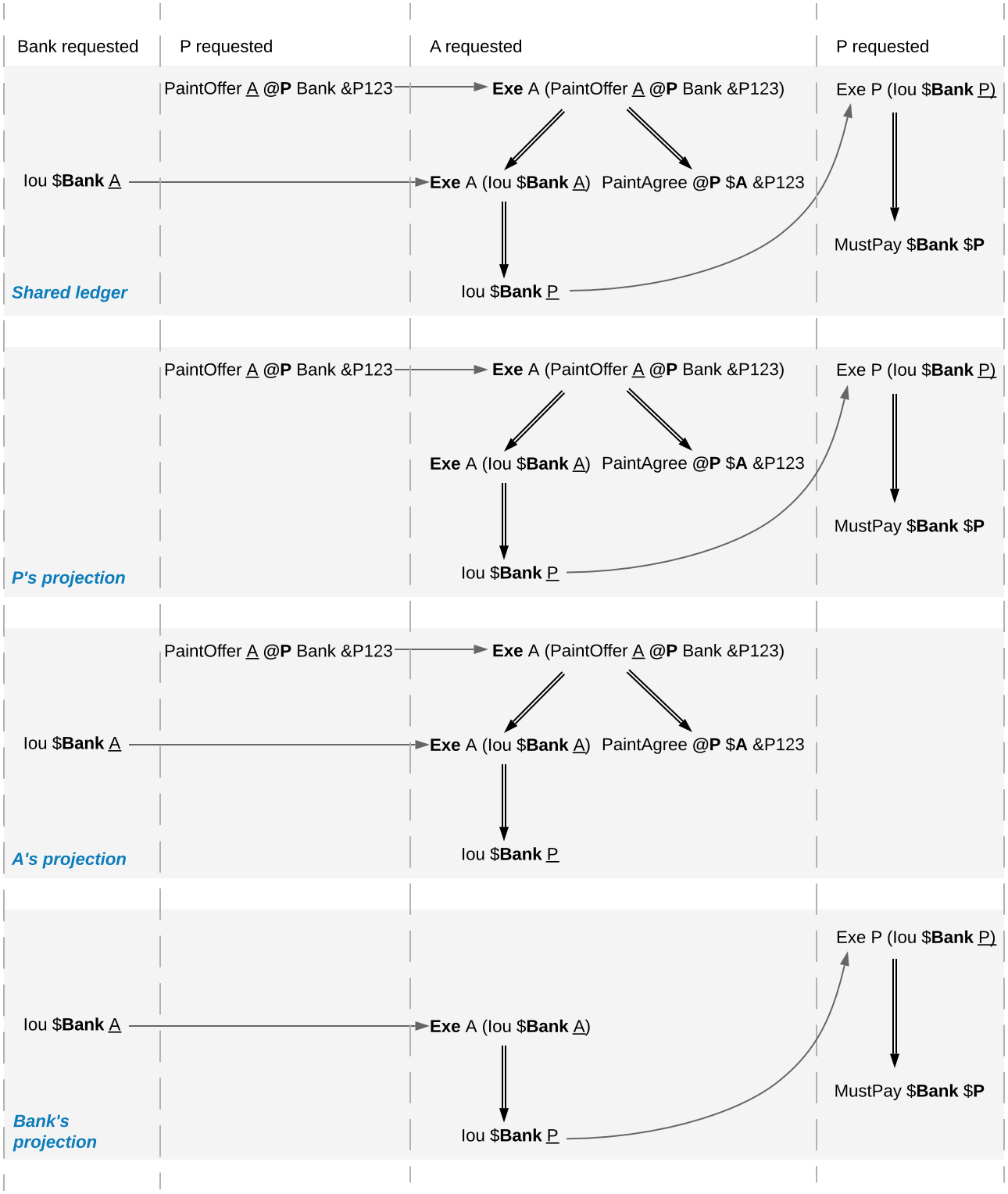
Showing contracts to non-stakeholders through ledger projections is called **divulgence**. Divulgence is a deliberate choice in the design of Daml ledgers. In the paint offer example, the only proper way to accept the offer is to transfer the money from Alice to the painter. Conceptually, at the instant where the offer is accepted, its stakeholders also gain a temporary stake in the actions on the two *Iou* contracts, even though they are never recorded as stakeholders in the contract model. Thus, they are allowed to see these actions through the projections.

More precisely, every action *act* on *c* is shown to all informees of all ancestor actions of *act*. These informees are called the **witnesses** of *act*. If one of the witnesses *W* is not a stakeholder on *c*, then *act* and *c* are said to be **divulged** to *W*. Note that only **Exercise** actions can be ancestors of other actions.

Divulgence can be used to enable delegation. For example, consider the scenario where Alice makes a counteroffer to the painter. Painter's acceptance entails transferring the IOU to him. To be able to construct the acceptance transaction, the painter first needs to learn about the details of the IOU that will be transferred to him. To give him these details, Alice can fetch the IOU in a context visible to the painter:



In the example, the context is provided by consuming a *ShowIou* contract on which the painter is a stakeholder. This now requires an additional contract type, compared to the original paint offer example. An alternative approach to enable this workflow, without increasing the number of contracts required, is to replace the original *Iou* contract by one on which the painter is a contract observer. This would require extending the contract model with a (consuming) exercise action on the *Iou* that creates a new *Iou*, with observers of Alice's choice. In addition to the different number of commits, the two approaches differ in one more aspect. Unlike stakeholders, parties who see contracts only through divulgence have no guarantees about the state of the contracts in question. For example, consider what happens if we extend our (original) paint offer example such that the painter immediately settles the IOU.



While Alice sees the creation of the *Iou Bank P* contract, she does not see the settlement action. Thus, she does not know whether the contract is still active at any point after its creation. Similarly, in the previous example with the counteroffer, Alice could spend the IOU that she showed to the painter by the time the painter attempts to accept her counteroffer. In this case, the painter's transaction could not be added to the ledger, as it would result in a double spend and violate validity. But the painter has no way to predict whether his acceptance can be added to the ledger or not.

1.44.1.5 Daml: Define Contract Models Compactly

As described in preceding sections, both the integrity and privacy notions depend on a contract model, and such a model must specify:

1. a set of allowed actions on the contracts, and
2. the signatories, contract observers, and
3. an optional agreement text associated with each contract, and
4. the optional key associated with each contract and its maintainers.

The sets of allowed actions can in general be infinite. For instance, the actions in the IOU contract model considered earlier can be instantiated for an arbitrary obligor and an arbitrary owner. As enumerating all possible actions from an infinite set is infeasible, a more compact way of representing models is needed.

Daml provides exactly that: a compact representation of a contract model. Intuitively, the allowed actions are:

1. **Create** actions on all instances of templates such that the template arguments satisfy the *ensure* clause of the template
2. **Exercise** actions on a contract corresponding to choices on that template, with given choice arguments, such that:
   1. The actors match the controllers of the choice. That is, the controllers define the *required authorizers* of the choice.
   2. The choice observers match the observers annotated in the choice.
   3. The exercise kind matches.
   4. All assertions in the update block hold for the given choice arguments.
   5. Create, exercise, fetch and key statements in the update block are represented as create, exercise and fetch actions and key assertions in the consequences of the exercise action.
3. **Fetch** actions on a contract corresponding to a *fetch* of that instance inside of an update block. The actors must be a non-empty subset of the contract stakeholders. The actors are determined dynamically as follows: if the fetch appears in an update block of a choice *ch* on a contract *c1*, and the fetched contract ID resolves to a contract *c2*, then the actors are defined as the intersection of (1) the signatories of *c1* union the controllers of *ch* with (2) the stakeholders of *c2*.
   A *fetchByKey* statement also produces a **Fetch** action with the actors determined in the same way. A *lookupByKey* statement that finds a contract also translates into a **Fetch** action, but all maintainers of the key are the actors.
4. **NoSuchKey** assertions corresponding to a *lookupByKey* update statement for the given key that does not find a contract.

An instance of a Daml template, that is, a **Daml contract**, is a triple of:

1. a contract identifier
2. the template identifier
3. the template arguments

The signatories of a Daml contract are derived from the template arguments and the explicit signatory annotations on the contract template. The contract observers are also derived from the template arguments and include:

1. the observers as explicitly annotated on the template
2. all controllers *c* of every choice defined using the syntax `controller c can...` (as opposed to the syntax `choice ... controller c`)

For example, the following template exactly describes the contract model of a simple IOU with a unit amount, shown earlier.

```
template MustPay with
    obligor : Party
    owner : Party
  where
    signatory obligor, owner
    agreement
      show obligor <> " must pay " <>
      show owner <> " one unit of value"

template Iou with
    obligor : Party
    owner : Party
  where
    signatory obligor
    observer owner

    choice Transfer
      : ContractId Iou
      with newOwner : Party
      controller owner
      do create Iou with obligor; owner = newOwner

    choice Settle
      : ContractId MustPay
      controller owner
      do create MustPay with obligor; owner
```

In this example, the owner is specified as an observer, since it must be able to see the contract to exercise the `Transfer` and `Settle` choices on it.

The template identifiers of contracts are created through a content-addressing scheme. This means every contract is self-describing in a sense: it constrains its stakeholder annotations and all Daml-conformant actions on itself. As a consequence, one can talk about the Daml contract model, as a single contract model encoding all possible instances of all possible templates. This model is subaction-closed; all exercise and create actions done within an update block are also always permissible as top-level actions.

1.44.1.6 Exceptions

The introduction of exceptions, a new Daml feature, has many implications for the ledger model. This page describes the changes to the ledger model introduced as part of this new feature.

Structure

Under the new feature, Daml programs can raise and catch exceptions. When an exception is caught in a *catch* block, the subtransaction starting at the corresponding *try* block is rolled back.

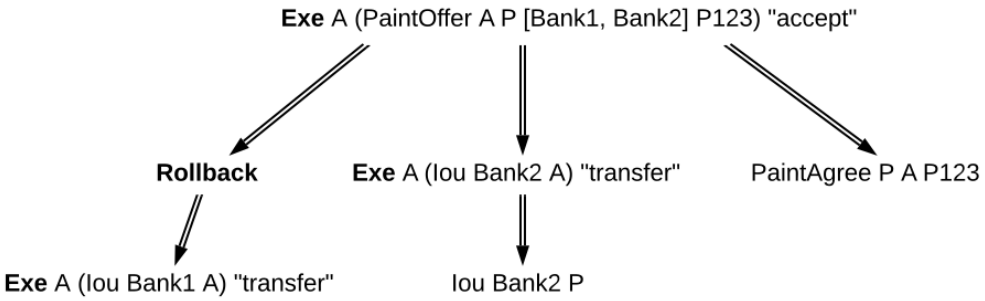
To support this in our ledger model, we need to modify the transaction structure to indicate which subtransactions were rolled back. We do this by introducing **rollback nodes** in the transaction. Each rollback node contains a rolled back subtransaction. Rollback nodes are not considered ledger actions.

Therefore we define transactions as a list of **nodes**, where each node is either a ledger action or a rollback node. This is reflected in the updated EBNF grammar for the transaction structure:

```
Transaction  ::= Node*
Node         ::= Action | Rollback
Rollback     ::= 'Rollback' Transaction
Action       ::= 'Create' contract
               | 'Exercise' party* contract Kind Transaction
               | 'Fetch' party* contract
               | 'NoSuchKey' key
Kind         ::= 'Consuming' | 'NonConsuming'
```

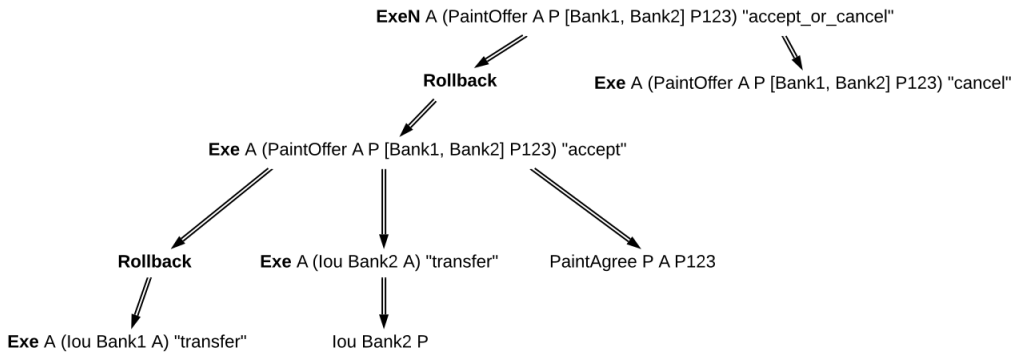
Note that *Action* and *Kind* have the same definition as before, but since *Transaction* may now contain rollback nodes, this means that an *Exercise* action may have a rollback node as one of its consequences.

For example, the following transaction contains a rollback node inside an exercise. It represents a paint offer involving multiple banks. The painter P is offering to paint A's house as long as they receive an Iou from Bank1 or, failing that, from Bank2. When A accepts, they confirm that transfer of an Iou via Bank1 has failed for some reason, so they roll it back and proceed with a transfer via Bank2:



Note also that rollback nodes may be nested, which represents a situation where multiple exceptions are raised and caught within the same transaction.

For example, the following transaction contains the previous one under a rollback node. It represents a case where the accept has failed at the last moment, for some reason, and a cancel exercise has been issued in response.



Consistency

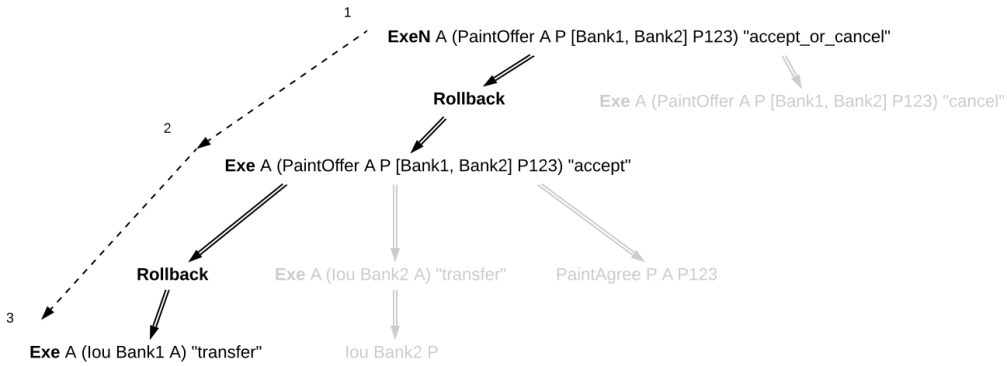
In the previous section on *consistency*, we defined a before-after relation on ledger actions. This notion needs to be revised in the presence of rollback nodes. It is no longer enough to perform a preorder traversal of the transaction tree, because the actions under a rollback node cannot affect actions that appear later in the transaction tree.

For example, a contract may be consumed by an exercise under a rollback node, and immediately again after the rollback node. This is allowed because the exercise was rolled back, and this does not represent a double spend of the same contract. You can see this in the nested example above, where the PaintOffer contract is consumed by an agree exercise, which is rolled back, and then by a cancel exercise.

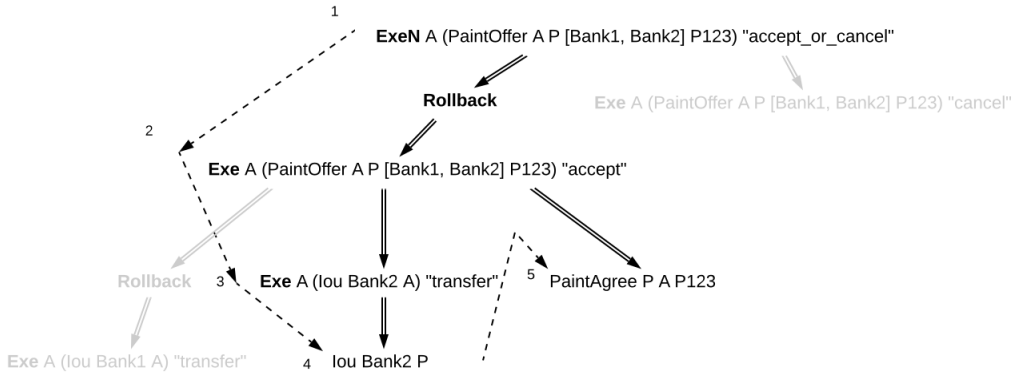
So, we now define the before-after relation as a partial order, rather than a total order, on all the actions of a transaction. This relation is defined as follows: *act1* comes before *act2* (equivalently, *act2* comes after *act1*) if and only if *act1* appears before *act2* in a preorder traversal of the transaction tree, and any rollback nodes that are ancestors of *act1* are also ancestors of *act2*.

With this modified before-after relation, the notion of internal consistency remains the same. Meaning that, for example, for any contract *c*, we still forbid the creation of *c* coming after any action on *c*, and we forbid any action on *c* coming after a consuming exercise on *c*.

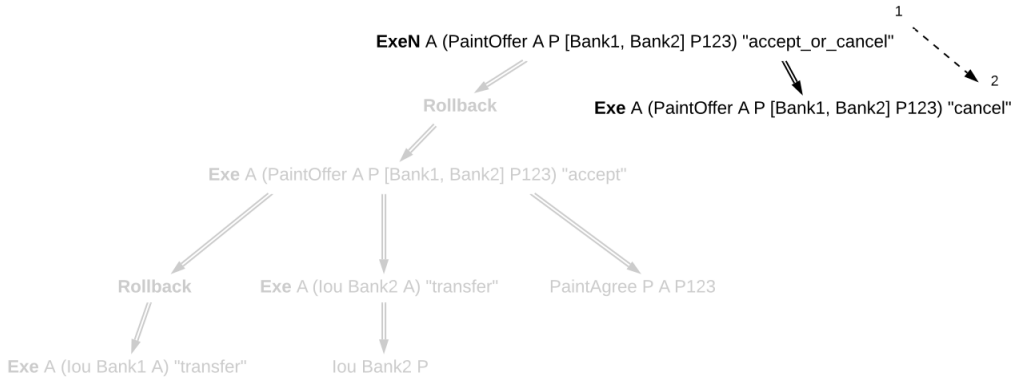
In the example above, neither consuming exercise comes after the other. They are part of separate continuities , so they don't introduce inconsistency. Here are three continuities implied by the before-after relation. The first:



The second:



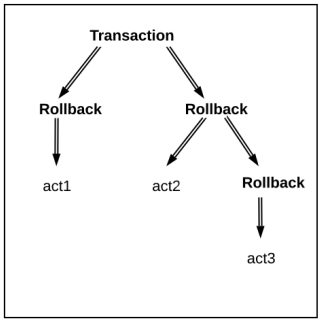
And the third:

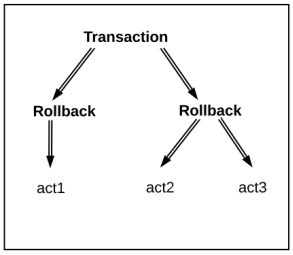


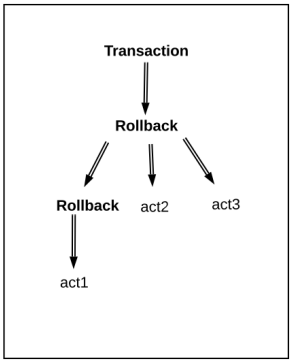
As you can see, in each of these continuities, no contract was consumed twice.

Transaction Normalization

The same before-after relation can be represented in more than one way using rollback nodes. For example, the following three transactions have the same before-after relation among their ledger actions (*act1*, *act2*, and *act3*):







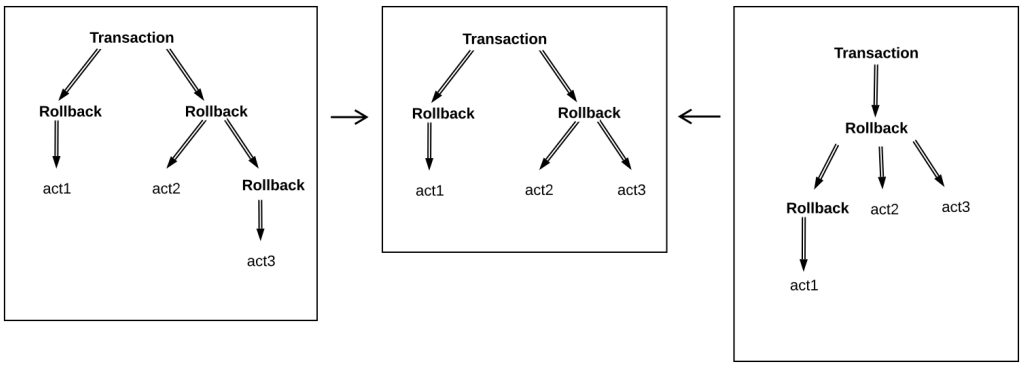
Because of this, these three transactions are equivalent. More generally, two transactions are equivalent if:

> The transactions are the same when you ignore all rollback nodes. That is, if you remove every rollback node and absorb its children into its parent, then two transactions are the same.

Equivalently, the transactions have the same ledger actions with the same preorder traversal and subaction relation.
The transactions have the same before-after relation between their actions.
The transactions have the same set of rollback children . A rollback child is an action whose direct parent is a rollback node.

For all three transactions above, the transaction tree ignoring rollbacks consists only of top-level actions (*act1*, *act2*, and *act3*), the before-after relation only says that *act2* comes before *act3*, and all three actions are rollback children. Thus all three transactions are equivalent.

Transaction normalization is the process by which equivalent transactions are converted into the same transaction. In the case above, all three transactions become the transaction in the middle when normalized.



To normalize a transaction, we apply three rules repeatedly across the whole transaction:

1. If a rollback node is empty, we drop it.
2. If a rollback node starts with another rollback node, for instance:

   ```
   'Rollback' [ 'Rollback' tx , node1, ..., nodeN ]
   ```

   Then we re-associate the rollback nodes, bringing the inner rollback node out:

   ```
   'Rollback' tx, 'Rollback' [ node1, ..., nodeN ]
   ```

3. If a rollback node ends with another rollback node, for instance:

   ```
   'Rollback' [ node1, ..., nodeN, 'Rollback' [ node1', ..., nodeM' ] ]
   ```

   Then we flatten the inner rollback node into its parent:

   ```
   'Rollback' [ node1, ..., nodeN, node1', ..., nodeM' ]
   ```

In the example above, using rule 3 we can turn the left transaction into the middle transaction, and using rule 2 we can turn the right transaction into the middle transaction. None of these rules apply to the middle transaction, so it is already normalized.

In the end, a normalized transaction cannot contain any rollback node that starts or ends with another rollback node, nor may it contain any empty rollback nodes. The normalization process minimizes the number of rollback nodes and their depth needed to represent the transaction.

To reduce the potential for information leaks, the ledger model must only contain normalized transactions. This also applies to projected transactions. An unnormalized transaction is always invalid.

Authorization

Since they are not ledger actions, rollback nodes do not have authorizers directly. Instead, a ledger is well-authorized exactly when the same ledger with rollback nodes removed (that is, replacing the rollback nodes with their children) is well-authorized, according to *the old definition*.

This is captured in the following rules:

When a rollback node is authorized by *p*, then all of its children are authorized by *p*. In particular:
- Top-level rollback nodes share the authorization of the requestors of the commit with all of its children.
- Rollback nodes that are a consequence of an exercise action *act* on a contract *c* share the authorization of the signatories of *c* and the actors of *act* with all of its children.
- A nested rollback node shares the authorization it got from its parent with all of its children.

The required authorizers of a rollback node are the union of all the required authorizers of its children.

Privacy

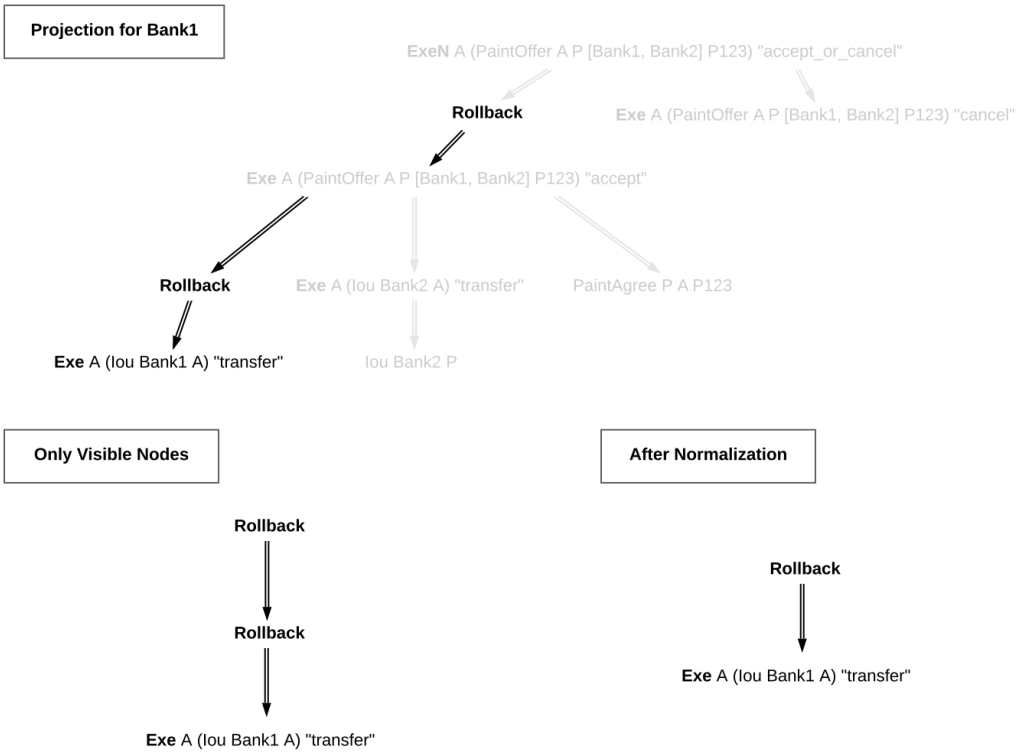
Rollback nodes also have an interesting effect on the notion of privacy in the ledger model. When projecting a transaction for a party *p*, it's necessary to preserve some of the rollback structure of the transaction, even if *p* does not have the right to observe every action under it. For example, we need *p* to be able to verify that a rolled back exercise (to which they are an informee) is conformant, but we also need *p* to know that the exercise was rolled back.

We adjust the definition of projection as follows:

1. For a ledger action, the projection for *p* is the same as it was before. That is, if *p* is an informee of the action, then the entire subtree is preserved. Otherwise the action is dropped, and the action's consequences are projected for *p*.
2. For a rollback node, the projection for *p* consists of the projection for *p* of its children, wrapped up in a new rollback node. In other words, projection happens under the rollback node, but the node is preserved.

After applying this process, the transaction must be normalized.

Consider the deeply nested example from before. To calculate the projection for Bank1, we note that the only visible action is the bottom left exercise. Removing the actions that Bank1 isn't an informee of, this results in a transaction containing a rollback node, containing a rollback node, containing an exercise. After normalization, this becomes a simple rollback node containing an exercise. See below:



The privacy section of the ledger model makes a point of saying that a contract model should be **subaction-closed** to support projections. But this requirement is not necessarily true once we introduce rollbacks. Rollback nodes may contain actions that are not valid as standalone actions, since they may have been interrupted prematurely by an exception.

Instead, we require that the contract model be **projection-closed**, i.e. closed under projections for any party 'p'. This is a weaker requirement that matches what we actually need.

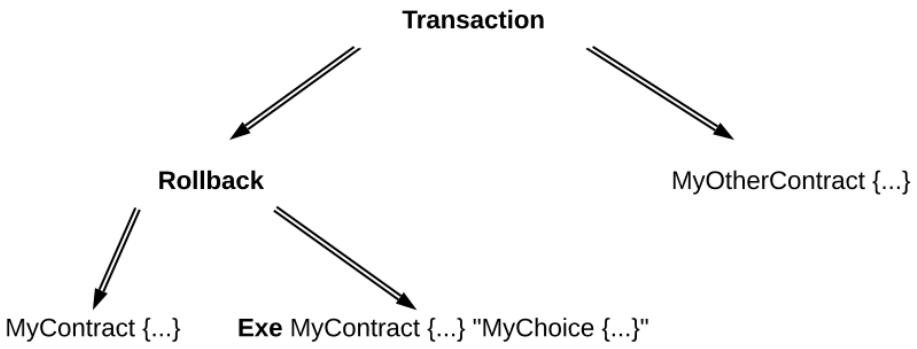
Relation to Daml Exceptions

Rollback nodes are created when an exception is thrown and caught within the same transaction. In particular, any exception that is caught within a try-catch will generate a rollback node if there are any ledger actions to roll back. For example:

```
try do
  cid <- create MyContract { ... }
  exercise cid MyChoice { ... }
  throw MyException
catch
  MyException ->
    create MyOtherContract { ... }
```

This Daml code will try to create a contract, and exercise a choice on this contract, before throwing an exception. That exception is caught immediately, and then another contract is created.

Thus a rollback node is created, to reset the ledger to the state it had at the start of the `try` block. The rollback node contains the create and exercise nodes. After the rollback node, another contract is created. Thus the final transaction looks like this:



Note that rollback nodes are only created if an exception is *caught*. An uncaught exception will result in an error, not a transaction.

After execution of the Daml code, the generated transaction is normalized.

1.44.1.7 Identity and Package Management

Since Daml ledgers enable parties to automate the management of their rights and obligations through smart contract code, they also have to provide party and code management functions. Hence, this document addresses:

1. Management of parties' digital identifiers in a Daml ledger.
2. Distribution of smart contract code between the parties connected to the same Daml ledger.

The access to this functionality is usually more restricted compared to the other Ledger API services, as they are part of the administrative API. This document is intended for the users and implementers of this API.

The administrative part of the Ledger API provides both a *party management service* and a *package service*. Any implementation of the party and package services is guaranteed to accept inputs and provide outputs of the format specified by these services. However, the services' *behavior* – the relationship between the inputs and outputs that the various parties observe – is largely implementation dependent. The remainder of the document will present:

1. The minimal behavioral guarantees for identity and package services across all ledger implementations. The service users can rely on these guarantees, and the implementers must ensure that they hold.
2. Guidelines for service users, explaining how different ledgers handle the unspecified part of the behavior.

Identity Management

A Daml ledger may freely define its own format of party and participant node identifiers, with some minor constraints on the identifiers' serialized form. For example, a ledger may use human-readable strings as identifiers, such as `Alice` or `Alice's Bank`. A different ledger might use public keys as identifiers, or the keys' fingerprints. The applications should thus not rely on the format of the identifier – even a software upgrade of a Daml ledger may introduce a new format.

By definition, identifiers identify parties, and are thus unique for a ledger. They do not, however, have to be unique across different ledgers. That is, two identical identifiers in two different ledgers

do not necessarily identify the same real-world party. Moreover, a real-world entity can have multiple identifiers (and thus parties) within the same ledger.

Since the identifiers might be difficult to interpret and manage for humans, the ledger may also accompany each identifier with a user-friendly **display name**. Unlike the identifier, the display name is not guaranteed to be unique, and two different participant nodes might return different display names for the same party identifier. Furthermore, a display name is in general not guaranteed to have any link to real world identities. For example, a party with a display name Attorney of Nigerian Prince might well be controlled by a real-world entity without a bar exam. However, particular ledger deployments might make stronger guarantees about this link. Finally, the association of identifiers to display names may change over time. For example, a party might change its display name from Bruce to Caitlyn – as long as the identifier remains the same, so does the party.

Provisioning Identifiers

The set of parties of any Daml ledger is dynamic: new parties may always be added to the system. The first step in adding a new party to the ledger is to provision a new identifier for the party. The Ledger API provides an *AllocateParty* method for this purpose. The method, if successful, returns an new party identifier. The `AllocateParty` call can take the desired identifier and display name as optional parameters, but these are merely hints and the ledger implementation may completely ignore them.

If the call returns a new identifier, the participant node serving this call is ready to host the party with this identifier. For some ledgers (Daml for VMware Blockchain in particular), the returned identifier is guaranteed to be **unique** in the ledger; namely, no other call of the `AllocateParty` method at this or any other ledger participant may return the same identifier. On Canton ledgers, the identifier is also unique as long as the participant node is configured correctly (in particular, it does not share its private key with other participant nodes).

After an identifier is returned, the ledger is set up in such a way that the participant node serving the call is allowed to issue commands and receive transactions on behalf of the party. However, the newly provisioned identifier need not be visible to the other participant nodes. For example, consider the setup with two participants `P1` and `P2`, where the party `Alice_123` is hosted on `P1`. Assume that a new party `Bob_456` is next successfully allocated on `P2`. As long as `P1` and `P2` are connected to the same Canton domain or Daml ledger, `Alice_123` can now submit a command with `Bob_456` as an informee.

For diagnostics, the ledger provides a *ListKnownParties* method which lists parties known to the participant node. The parties can be local (i.e., hosted by the participant) or not.

Identifiers and Authorization

To issue commands or receive transactions on behalf of a newly provisioned party, an application must provide a proof to the party's hosting participant that they are authorized to represent the party. Before the newly provisioned party can be used, the application will have to obtain a token for this party. The issuance of tokens is specific to each ledger and independent of the Ledger API. The same is true for the policy which the participants use to decide whether to accept a token.

To learn more about Ledger API security model, please read the *Authorization documentation*.

Identifiers and the Real World

The substrate on which Daml workflows are built are the real-world obligations of the parties in the workflow. To give value to these obligations, they must be connected to parties in the real world. However, the process of linking party identifiers to real-world entities is left to the ledger implementation.

In centralized deployments, one can simplify the process by trusting the operator of the writer node(s) with providing the link to the real world. For example, if the operator is a stock exchange, it might guarantee that a real-world exchange participant whose legal name is Bank Inc. is represented by a ledger party with the identifier Bank Inc. . Alternatively, it might use a random identifier, but guarantee that the display name is Bank Inc. . In general, a ledger might not have such a single store of identities. The solutions for linking the identifiers to real-world identities could rely on certificate chains, verifiable credentials, or other mechanisms. The mechanisms can be implemented off-ledger, using Daml workflows (for instance, a know your customer workflow), or a combination of these.

Package Management

All Daml ledgers implement endpoints that allow for provisioning new Daml code to the ledger. The vetting process for this code, however, depends on the particular ledger implementation and its configuration. The remainder of this section describes the endpoints and general principles behind the vetting process. The details of the process are ledger-dependent.

Package Formats and Identifiers

Any code – i.e., Daml templates – to be uploaded must compiled down to the *Daml-LF* language. The unit of packaging for Daml-LF is the *.dalf* file. Each `.dalf` file is uniquely identified by its **package identifier**, which is the hash of its contents. Templates in a `.dalf` file can reference templates from other `.dalf` files, i.e., `.dalf` files can depend on other `.dalf` files. A *.dar* file is a simple archive containing multiple `.dalf` files, and has no identifier of its own. The archive provides a convenient way to package `.dalf` files together with their dependencies. The Ledger API supports only `.dar` file uploads. Internally, the ledger implementation need not (and often will not) store the uploaded `.dar` files, but only the contained `.dalf` files.

Package Management API

The package management API supports two methods:

UploadDarFile for uploading `.dar` files. The ledger implementation is, however, free to reject any and all packages and return an error. Furthermore, even if the method call succeeds, the ledger's vetting process might restrict the usability of the template. For example, assume that Alice successfully uploads a `.dar` file to her participant containing a `NewTemplate` template. It may happen that she can now issue commands that create `NewTemplate` instances with Bob as a stakeholder, but that all commands that create `NewTemplate` instances with Charlie as a stakeholder fail.

ListKnownPackages that lists the `.dalf` package vetted for usage at the participant node. Like with the previous method, the usability of the listed templates depends on the ledger's vetting process.

Package Vetting

Using a Daml package entails running its Daml code. The Daml interpreter ensures that the Daml code cannot interact with the environment of the system on which it is executing. However, the operators of the ledger infrastructure nodes may still wish to review and vet any Daml code before allowing it to execute. One reason for this is that the Daml interpreter currently lacks a notion of reproducible resource limits, and executing a Daml contract might result in high memory or CPU usage.

Thus, Daml ledgers generally allow some form of vetting a package before running its code on a node. Not all nodes in a Daml ledger must vet all packages, as it is possible that some of them will not execute the code. The exact vetting mechanism is ledger-dependent. For example, in the *Daml Sandbox*, the vetting is implicit: uploading a package through the Ledger API already vets the package, since it's assumed that only the system administrator has access to these API facilities. The vetting process can be manual, where an administrator inspects each package, or it can be automated, for example, by accepting only packages with a digital signature from a trusted package issuer.

In Canton, participant nodes also only need to vet code for the contracts of the parties they host. As only participants execute contract code, only they need to vet it. The vetting results may also differ at different participants. For example, participants `P1` and `P2` might vet a package containing a `NewTemplate` template, whereas `P3` might reject it. In that case, if Alice is hosted at `P1`, she can create `NewTemplate` instances with stakeholder Bob who is hosted at `P2`, but not with stakeholder Charlie if he's hosted at `P3`.

Package Upgrades

The Ledger API does not have any special support for package upgrades. A new version of an existing package is treated the same as a completely new package, and undergoes the same vetting process. Upgrades to active contracts can be done by the Daml code of the new package version, by archiving the old contracts and creating new ones.

1.44.1.8 Time on Daml Ledgers

The Daml language contains a function *getTime* which returns the current time . However, the concept of a current time can be challenging in a distributed setting.

This document describes the detailed semantics of time on Daml ledgers, centered around the two timestamps assigned to each transaction: the *ledger time* `lt_TX` and the *record time* `rt_TX`.

Ledger Time

The *ledger time* `lt_TX` is a property of a transaction. It is a timestamp that defines the value of all *getTime* calls in the given transaction, and has microsecond resolution. The ledger time is assigned by the submitting participant as part of the Daml command interpretation.

Record Time

The *record time* `rt_TX` is another property of a transaction. It is a timestamp with microsecond resolution, and it is assigned by the backing storage mechanism when the transaction is persisted.

The record time should be an intuitive representation of real time , but the Daml abstract ledger model does not prescribe exactly how to assign the record time. Each persistence technology might use a different way of representing time in a distributed setting.

Guarantees

The ledger time of a valid transaction `TX` must fulfill the following rules:

1. **Causal monotonicity**: for any action (create, exercise, fetch, lookup) in `TX` on a contract `C`, `lt_TX >= lt_C`, where `lt_C` is the ledger time of the transaction that created `C`.
2. **Bounded skew**: `rt_TX - skew_min <= lt_TX <= rt_TX + skew_max`, where `skew_min` and `skew_max` are parameters defined by the ledger.

Apart from that, no other guarantees are given on the ledger time. In particular, neither the ledger time nor the record time need to be monotonically increasing.

Time has therefore to be considered slightly fuzzy in Daml, with the fuzziness depending on the skew parameters. Daml applications should not interpret the value returned by *getTime* as a precise timestamp.

Ledger Time Model

The *ledger time model* is the set of parameters used in the assignment and validation of ledger time. It consists of the following:

1. `skew_min` and `skew_max`, the bounds on the difference between `lt_TX` and `rt_TX`.
2. `transaction_latency`, the average duration from the time a transaction is submitted from a participant to the ledger until the transaction is recorded. This value is used by the participant to account for latency when submitting transactions to the ledger: transactions are submitted slightly ahead of their ledger time, with the intention that they arrive at `lt_TX == rt_TX`.

The ledger time model is part of the ledger configuration and can be changed by ledger operators through the *SetTimeModel* config management API.

Assign Ledger Time

The ledger time is assigned automatically by the participant. In most cases, Daml applications will not need to worry about ledger time and record time at all.

For reference, this section describes the details of how the ledger time is currently assigned. The algorithm is not part of the definition of time in Daml, and may change in the future.

1. When submitting commands over the ledger API, users can optionally specify a `min_ledger_time_rel` or `min_ledger_time_abs` argument. This defines a lower bound for the ledger time in relative and absolute terms, respectively.
2. The ledger time is set to the highest of the following values:

    1. `max(lt_C_1, ..., lt_C_n)`, the maximum ledger time of all contracts used by the given transaction
    2. `t_p`, the local time on the participant
    3. `t_p + min_ledger_time_rel`, if `min_ledger_time_rel` is given
    4. `min_ledger_time_abs`, if `min_ledger_time_abs` is given
3. Since the set of commands used by a given transaction can depend on the chosen time, the above process might need to be repeated until a suitable ledger time is found.
4. If no suitable ledger time is found after 3 iterations, the submission is rejected. This can happen if there is contention around a contract, or if the transaction uses a very fine-grained control flow based on time.
5. At this point, the ledger time may lie in the future (e.g., if a large value for `min_ledger_time_rel` was given). The participant waits until `lt_TX - transaction_latency` before it submits the transaction to the ledger - the intention is that the transaction is recorded at `lt_TX == rt_TX`.

Use the parameters `min_ledger_time_rel` and `min_ledger_time_abs` if you expect that command interpretation will take a considerate amount of time, such that by the time the resulting transaction is submitted to the ledger, its assigned ledger time is not valid anymore. Note that these parameters can only make sure that the transaction arrives roughly at `rt_TX` at the ledger. If a subsequent validation on the ledger takes longer than `skew_max`, the transaction will still be rejected and you'll have to ask your ledger operator to increase the `skew_max` time model parameter.

1.44.2 Canton Advanced Architecture

1.44.2.1 Contract Keys in Canton

Daml provides a contract key mechanism for contracts, similar to primary keys in relational databases. When using multi-domain topologies, Canton will support the full syntax of contract keys, but only a reduced semantics. That is, all valid Daml contracts using keys will run on Canton, but their behavior may deviate from the prescribed one. This document explains the deviation, as well as ways of recovering the full functionality of keys in some scenarios. It assumes a reasonable familiarity with Daml.

Note: This section covers a preview feature, when using contract keys in a multi-domain setup. By default, contract key uniqueness is enabled, and therefore this section does not apply. However, contract key uniqueness will soon be deprecated, as uniqueness cannot be enforced among multiple domains. We encourage to build your models already anticipating this change.

Keys have two main functions:

Simplifying the modeling of mutable state in Daml. Daml contracts are immutable and can be only created and archived. Mutating a contract `C` is modeled by archiving `C` and creating a new contract `C'` which is a modified version of `C`. Other than keys, Daml offers no means to capture the relation between `C` and `C'`. After archiving `C`, any contract `D` that contains the contract ID of `C` is left with a dangling reference. This makes it cumbersome to model mutable state that is split across multiple contracts. Keys provide mutable references in Daml; giving `C` and `C'` the same key `K` allows `D` to store `K` as a reference that will start pointing to `C'` after archiving `C`.

Checking that no active contract with a given key exists at some point in time. This mainly serves to provide uniqueness guarantees, which are useful in many cases.

One is that they can serve to de-duplicate data coming from external sources. Another one is that they allow natural mutable references, e.g., referring to a user by their username or e-mail.

Canton participants and domains can be run in two modes:

1. **In *unique-contract-key (UCK) mode*, contract keys in Canton provide both functions; there can be at mo**
   However, only UCK participants can connect to UCK domains and the first UCK domain a UCK participant connects to is the only domain that the participant can connect to in its lifetime. UCK domains and their participants are thus isolated islands that are deprived of Canton's composability and interoperability features.
2. In *non-unique-keys mode*, contract keys in Canton provide the first, but not the second function, at least not without additional effort or restrictions. In particular:
   1. In Canton, two (or more) active contracts with the same key may exist simultaneously on the same or different domains.
   2. If no submitting party is a stakeholder of an active contract instance of template `Template` with the key `k` visible on the submitting participant when the participant processes the submission, then a `lookupByKey @Template k` may return `None` even if an active contract instance of template `Template` with the key `k` exists on the virtual shared ledger at the point in time when the transaction is committed.
   3. A `fetchByKey @Template k` or an `exerciseByKey @Template k` or a positive `lookupByKey @Template k` (returning `Some cid`) may return any active contract of template `Template` with key `k`.

In the remainder of the document we:

give *more detailed examples* of the differences above
give an *overview of how keys are implemented* so that you can better understand their behavior
show *workarounds for recovering the uniqueness functionality* in particular scenarios on normal domains
give a *formal semantics of keys* in Canton, in terms of the *Daml ledger model*
explain how to *run a domain in UCK mode*.

Domains with Uniqueness Guarantees

By default, Canton domains and participants are currently configured to provide unique contract key (UCK) semantics. This will be deprecated in the future, as such a uniqueness constraint cannot be supported on a distributed system in a useful way. The *semantic differences from the ledger model* disappear if the transactions are submitted to a participant connected to a Canton domain in *UCK mode*. The *workarounds* are therefore not needed.

A UCK participant can connect only to a UCK domain. Moreover, once it has successfully connected to a UCK domain, it will refuse to connect to another domain. Accordingly, conflict detection on a single domain suffices to check for key uniqueness. Participants connected to a UCK domain check for key conflicts whenever they host one of the key maintainers:

When a contract is created, they check that there is no other active contract with the same key.
When the submitted transaction contains a negative key lookup, the participants check that there is indeed no active contract for the given key.

Warning: Daml workflows deployed on a UCK domain are locked into this domain. They cannot use Canton's composability and interoperability features because the participants will refuse to connect to other domains.

Non Unique Contract Keys Mode

This section explains how contract keys behave on participants connected to Canton domains without unique contract keys. This mode can be activated by setting

```
canton {
    domains {
        alpha {
            // subsequent changes have no effect and the mode of a node can never
↪be changed
            init.domain-parameters.unique-contract-keys = false
        }
    }
    participants {
        participant1 {
            // subsequent changes have no effect and the mode of a node can never
↪be changed
            init.parameters.unique-contract-keys = false
        }
    }
}
```

Note: Non-Unique contract keys is preview only and currently broken. Multiple keys will override each other.

Examples of Semantic Differences

Double Key Creation

Consider the following template:

```
template Keyed
  with
    sig: Party
    k: Int
  where
    signatory sig
    key (sig, k): (Party, Int)
    maintainer key._1
```

The Daml contract key semantics prescribe that no two active `Keyed` contracts with the same keys should exist. For example, consider the following Daml script:

```
multiple = script do
  alice <- allocateParty "alice"
```

(continues on next page)

(continued from previous page)

```
  submitMustFail alice $ do
    createCmd (Keyed with sig = alice, k = 1)
    createCmd (Keyed with sig = alice, k = 1)
    pure ()
```

Alice's submission must fail, since it attempts to create two contracts with the key `(Alice, 1)`. In Canton, however, the submission is legal and will succeed (if executed, for example, through Daml Script). Thus, you cannot directly rely on keys to ensure the uniqueness of user-chosen usernames or external identifiers (e.g., order identifiers, health record identifiers, entity identifiers) in Canton.

False `lookupByKey` Negatives

Similarly, your code might rely on the negative case of a `lookupByKey`:

```
template Initialization
  with
    sig: Party
    k: Int
  where
    signatory sig

template Orchestrator
  with
    sig: Party
  where
    signatory sig

    nonconsuming choice Initialize: Optional (ContractId Initialization)
      with
        k: Int
      controller sig
        do
          optCid <- lookupByKey @Keyed (sig, k)
          case optCid of
            None -> do
              create Keyed with ..
              time <- getTime
              cid <- create Initialization with sig, k
              pure $ Some cid
            Some _ -> pure None
```

When running a process (represented by the `Initialization` template here), you might use a pattern like above to ensure that it is run only once. The `Initialization` template does not have a key. Nevertheless, if all processing happens through the `Orchestrator` template, there will only ever be one `Initialization` created for the given party and key. For example, the following script creates only one `Initialization` contract:

```
lookupNone = script do
  alice <- allocateParty "alice"
  orchestratorId <- submit alice do
    createCmd Orchestrator with sig = alice
  submit alice do
    exerciseCmd orchestratorId Initialize with k = 1
```

(continues on next page)

(continued from previous page)

```
  submit alice do
    exerciseCmd orchestratorId Initialize with k = 1
```

In scripts, transactions are executed sequentially. Alice's second submission above will always find the existing `Keyed` contract, and thus execute the `Some` branch of the `Initialize` choice. In real-world applications, transactions may run concurrently. Assume that `initTx1` and `initTx2` are run concurrently, and that these are the first two transactions running the `Initialize` choice. Then, during their preparation, both of them might execute the `None` branch (i.e., `lookupByKey` might return a negative result), and thus both might try to create the `Initialization` contract. However, negative `lookupByKey` results must be committed to the ledger, and the *key consistency requirements* prohibit both of them committing. Thus, one of `initTx1` and `initTx2` might fail, or they both might succeed (if one of them sees the effects of the other and then executes the `Some` branch), but in either case, only one `Initialization` contract will be created.

In Canton, however, it is possible that both `initTx1` and `initTx2` execute the `None` branch, yet both get committed. For example, if the participant processes the submissions for `initTx1` and `initTx2` concurrently, neither will see `initTx1` the `Initialization` contract created by `initTx2` nor vice versa. Canton orders the transactions only after the commands have been interpreted, and in normal mode it does not check the consistency of negative lookup by keys after ordering any more. Thus, two `Initialization` contracts may get created.

Semantics of `fetchByKey` and Positive `lookupByKey`

Daml also provides a `fetchByKey` operation. Daml commands are evaluated against some active contract set. When Daml encounters a `fetchByKey` command, it tries to find an active contract with the given key (and fails if it cannot). Since Daml semantics prescribe that only one such contract may exist, it is clear which one to return. For example, consider the script:

```
fetchSome = script do
  alice <- allocateParty "alice"
  keyedId1 <- submit alice do
    createCmd Keyed with sig = alice, k = 1
  keyedId2 <- submitMustFail alice do
    createCmd Keyed with sig = alice, k = 1
  (foundId, _) <- submit alice do
    createAndExerciseCmd (KeyedHelper alice) $ FetchByKey (alice, 1)
  assert $ foundId == keyedId1
  optFoundId <- submit alice do
    createAndExerciseCmd (KeyedHelper alice) $ LookupByKey (alice, 1)
  assert $ optFoundId == Some keyedId1
```

The script uses a helper template `KeyedHelper` shown at the end of this section because `fetchByKey` and `lookupByKey` cannot be used directly in a Daml script.

Daml's contract key semantics says that Alice's second submission must fail, since a contract with the given key already exists. Thus, her third submission will always succeed, and return `keyedId1`, since this is the only `Keyed` contract with the key `(Alice, 1)`. Similarly, her fourth submission will also successfully find a contract, which will be `keyedId1`.

As discussed earlier, Alice's second submission in the above script will succeed in Canton. Alice's third and fourth submissions thus may return different contract IDs, with each returning either `keyedId1`, or `keyedId2`. Whichever one is returned, a successful `fetchByKey` and `lookupByKey`

still guarantees that the returned contract is active at the time when the transaction gets committed. As mentioned earlier, negative `lookupByKey` results may be spurious.

```
template KeyedHelper
  with
    p: Party
  where
    signatory p

    choice FetchByKey: (ContractId Keyed, Keyed)
      with keyP: (Party, Int)
      controller p
      do fetchByKey @Keyed keyP

    choice LookupByKey: Optional (ContractId Keyed)
      with keyP: (Party, Int)
      controller p
      do lookupByKey @Keyed keyP
```

Canton's Implementation of Keys

Internally, a Canton participant node has a component that provides the gRPC interface (the Ledger API Server), and another component that synchronizes participants (the sync service). When a command is submitted, the Ledger API Server evaluates the command against its local view, including the resolution of key lookups (`lookupByKey` and `fetchByKey`). Submitted commands are evaluated in parallel, both on a single node and across different nodes.

The evaluated command is then sent to the sync service, which runs Canton's *commit protocol*. The protocol provides a linear ordering of all transactions on a single domain, and participants check all transactions for conflicts, with an earlier-transaction-wins policy. As participants only see parts of transactions (the joint *projection* of the parties they host), they only check conflicts on contracts for which they host stakeholders. During conflict detection, positive key lookups (that find a contract ID based on a key) are treated as ordinary `fetch` commands on the found contract ID, and the contract ID is checked to still be active. Negative key lookups, on the other hand, are never checked by Canton (a malicious submitter, for example, can always successfully claim that the lookup was negative). Similarly, contract creations are not checked for duplicate keys. Logically, both of these checks would require checking a there is no such key statement. Canton does not check such statements. While adding the check to the individual participants is straightforward, it is hard to get meaningful guarantees from such local checks because each participant has only a limited view of the entire virtual global ledger. For example, the check could pass locally on a participant even though there exists a contract with the given key on some domain that the participant is not connected to. Similarly, since the processing of different domains runs in parallel, it is unclear how to consistently handle the case where transactions on different domains create two contracts with the same key.

For integrity, the participants also re-evaluate the submitted command (or, more precisely, the subtransaction in the joint *projection* of the parties they host). The commit protocol ensures that any two involved participants will evaluate the key lookups in the same way as the Ledger API Server of the submitting participant. That is, if there are two active contracts with the key `k`, the protocol insures that a `fetchByKey k` will return the same contract on all participants.

Once the sync protocol commits a transaction, it informs the Ledger API server, which then atomically updates its set of active contracts. The transactions are passed to the Ledger API server in the order

in which they are recorded on the ledger.

Workarounds for Recovering Uniqueness

Since some form of uniqueness for ledger data is necessary in many cases, we list some strategies to achieve it in Canton without being locked into a UCK domain. The strategies' applicability depends on your contracts and the deployment setup of your application. In general, none of the strategies apply to the case where creations and deletions of contracts with keys are delegated.

Setting: Single Maintainer, Single Participant Node

Often, contracts may have a single maintainer (e.g., an operator that wants to have unique user names for its users). In the simplest case, the maintainer party will be hosted on just one participant node. This setting allows some simple options for recovering uniqueness.

Command ID Deduplication

The Ledger API server deduplicates commands based on their IDs. Note, however, that the IDs are deduplicated only within a configured window of time. This can simplify the uniqueness bookkeeping of your application as follows. Before your application sends a command that creates a contract with the key `k`, it should first check that no contract with the key `k` exists in a recent ACS snapshot (obtained from the Ledger API). Then, it should use a command ID that is a deterministic function of `k` to send the command. This protects you from the race condition of creating the key twice concurrently, without having to keep track of commands in flight. Caveats to keep in mind are:

- you need to know exactly which contracts with keys each of your commands will create
- your commands may only create contracts with a single key `k`
- only the maintainer party may submit commands that create contracts with keys (i.e., do not delegate the creation to other parties).

However, these conditions are often true in simple cases (e.g., commands that create new users).

Generator Contract

Another approach is to funnel all creations of the keyed contracts through a generator contract. An example generator for the `Keyed` template is shown below.

```
template Generator
  with
    sig: Party
  where
    signatory sig

    choice Generate : (ContractId Generator, ContractId Keyed)
      with
        k: Int
      controller sig
        do
          existing <- lookupByKey @Keyed (sig, k)
```

(continues on next page)

(continued from previous page)

```
          keyed <- case existing of
            Some cid -> pure cid
            None ->
              create Keyed with ..
          gen <- create this
          pure (gen, keyed)
```

The main difference from the `Orchestrator` contract is that the `Generate` choice is consuming. Caveats to keep in mind are:

Your application must ensure that you only ever create one `Generator` contract (e.g., by creating one when initializing the application for the first time).
All commands that create the `Keyed` contract must be issued by the maintainer (in particular, do not delegate choices on the `Generator` contract to other parties).
You must not create `Keyed` contracts by any other means other than exercising the `Generate` choice.
The `Generate` choice as shown above will not abort the command if the contract with the given key already exists, it will just return the existing contract. However, this is easy to change.
This approach relies on a particular internal behavior of Canton (as discussed below). While we don't expect the behavior to change, we do not currently make strong guarantees that it will not change.
If the participant is connected to multiple domains, the approach may fail in future versions of Canton. To be future-proof, you should only use it in the settings when your participant is connected to a single domain.

A usage example script is below.

```
generator = script do
  alice <- allocateParty "Alice"
  -- Your application must ensure that the following command runs at most once
  gen <- submit alice $
    createCmd Generator with sig = alice
  (gen, keyed) <- submit alice $
    exerciseCmd gen Generate with k = 1
  (gen, keyed1) <- submit alice $
    exerciseCmd gen Generate with k = 1
  assert $ keyed1 == keyed
  submit alice $
    exerciseCmd keyed Archive
  (gen, keyed2) <- submit alice $
    exerciseCmd gen Generate with k = 1
  assert $ keyed2 /= keyed
```

To understand why this works, first read how keys are *implemented in Canton*. With this in mind, since the `Generate` choice is consuming, if you issue two or more concurrent commands that use the `Generate` choice, at most one of them will succeed (as the `Generator` contract will be archived when the first transaction commits). Thus, all accepted commands will be evaluated sequentially by the Ledger API server. As the server writes the results of accepted commands to its database atomically, the `Keyed` contract created by one command that uses `Generate` will either be visible to the following command that uses `Generate`, or it will have been archived by some other, unrelated command in between.

Setting: Single Maintainer, Multiple Participants

Ensuring uniqueness with multiple participants is more complicated, and adds more restrictions on how you operate on the contract.

The main approach is to track all allocations and deallocations of a key through a helper contract.

```
template KeyState
  with
    sig: Party
    k: Int
    allocated: Bool
  where
    signatory sig

    choice Allocate : (ContractId KeyState, ContractId Keyed)
      controller sig
        do
          assert $ not allocated
          newState <- create this with allocated = True
          keyed <- create Keyed with ..
          pure (newState, keyed)

    choice Deallocate : ContractId KeyState
      controller sig
        do
          assert $ allocated
          (cid, _) <- fetchByKey @Keyed (sig, k)
          exercise cid Archive
          create this with allocated = False
```

Caveats:

> Before creating a contract with the key `k` for the first time, your application must create the matching `KeyState` contract with `allocated` set to `False`. Such a contract must be created at most once. Most likely, you will want to choose a master participant on which you create such contracts.
> Do not delegate choices on the `Keyed` contract to parties other than the maintainers.
> You must never send a command that creates or archives the `Keyed` contract directly. Instead, you must use the `Allocate` and `Deallocate` choices on the `KeyState` contract. The only exception are consuming choices on the `Keyed` contract that immediately recreate a `Keyed` contract with the same key. These choices may also be delegated.

A usage example script is below.

```
state = script do
  alice <- allocateParty "Alice"
  -- Your application must ensure that the following command executes at most once
  state <- submit alice $
    createCmd KeyState with sig = alice, k = 1, allocated = False
  (state, keyed) <- submit alice $
    exerciseCmd state Allocate
  submitMustFail alice $
    exerciseCmd state Allocate
  -- If you archive the keyed contract without going through the
```

(continues on next page)

(continued from previous page)

```
  -- KeyState, you must also recreate it in the same transaction.
  -- For example, if Keyed had consuming choices, the choices' bodies
  -- would have to recreate another Keyed contract with the same key
  submit alice $ do
    exerciseCmd keyed Archive
    createCmd Keyed with sig = alice, k = 1
    pure ()
  state <- submit alice $
    exerciseCmd state Deallocate
  (state, keyed2) <- submit alice $
    exerciseCmd state Allocate
  assert $ keyed2 /= keyed
```

An alternative to this approach, if you want to use a consuming choice `ch` on the `Keyed` template that doesn't recreate key, is to record the contract ID of the `KeyState` contract in the `Keyed` contract. You can then call `Deallocate` from `ch`, but you must first modify `Deallocate` to not perform a `lookupByKey`.

Setting: Multiple Maintainers

Achieving uniqueness for contracts with multiple maintainers is more difficult, and the maintainers must trust each other. To handle this case, follow the `KeyState` approach from the previous section. The main difference is that the `KeyState` contracts must have multiple signatories. Thus you must follow the usual Daml pattern of collecting signatories. Be aware that you must still structure this such that you only ever create one `KeyState` contract.

Formal Semantics of Keys in Canton

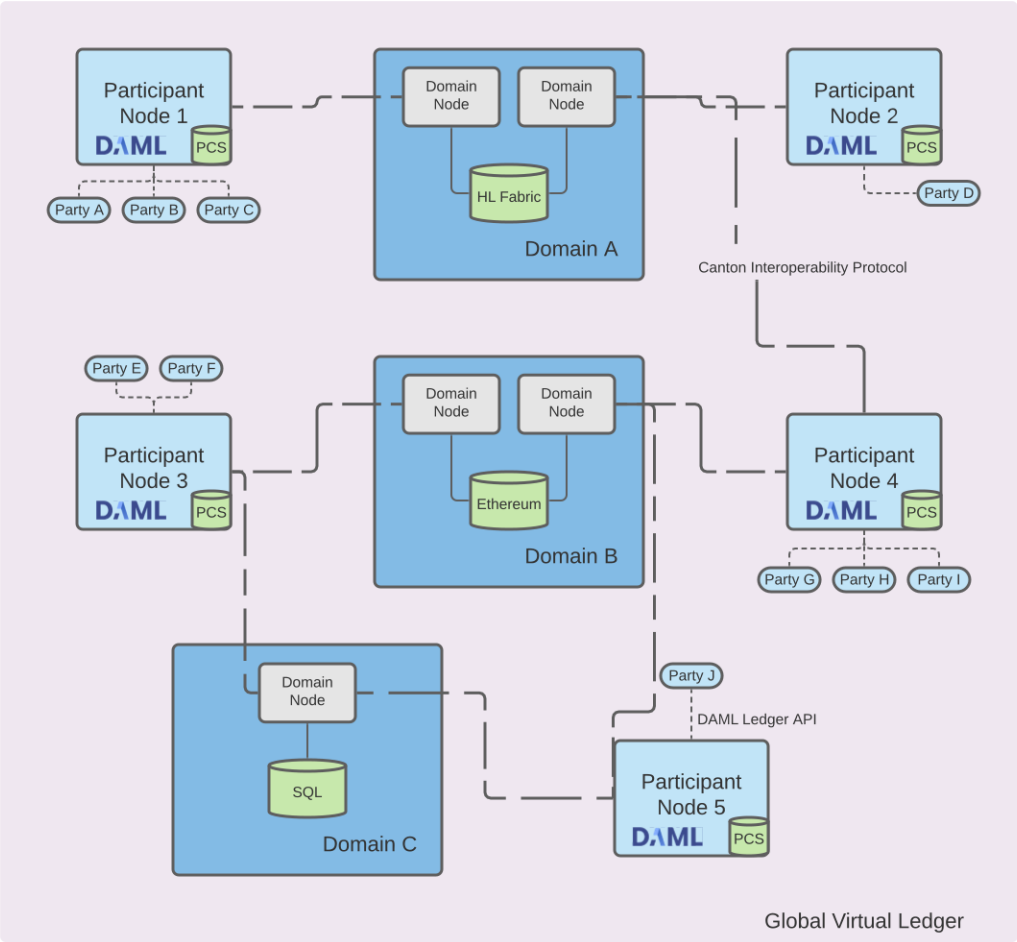
In terms of the *Daml ledger model*, Canton's virtual shared ledger satisfies key consistency only when it represents a single UCK domain. In general, Canton's virtual shared ledger violates key consistency. That is, `NoSuchKey k` actions may happen on the ledger even when there exists an active contract with the key `k`. Similarly, `Create` actions for a contract with the key `k` may appear on the ledger even if another active contract with the key `k` exists.

In terms of Daml evaluation, i.e., the translation of Daml into the ledger model transactions, the following changes:

> When evaluated against an active contract set, a `fetchByKey k` may result in a `Fetch c` action for any active contract `c` with the key `k` (in Canton, there can be multiple such contracts). In the current implementation, it will favor the most recently created contract within the single transaction. However, this is not guaranteed to hold in future versions of Canton. If no contract with key `k` is active, it will fail as usual.
> Similarly, `lookupByKey k` may result in a `Fetch c` for any active contract `c` with the key `k` of which the submitter is a stakeholder. If no such contract exists, it results in a `NoSuchKey k` as usual.
> Likewise, an `exerciseByKey k` may result in an `Exercise` on any contract `c` with the key `k`. It fails if no contract with key `k` is active.

1.44.2.2 Domain Architecture and Integrations

Recall the high-level topology with Canton domains being backed by different technologies, such as a relational database as well as block-chains like Hyperledger Fabric or Ethereum.



In this chapter we define the requirements specific to a Canton domain, explain the generic domain architecture, as well as the concrete integrations for Canton domains.

Domain-specific Requirements

The *high-level requirements* define requirements for Canton in general, covering both participant and domains. This section categorizes and expands on these high-level requirements and defines domain-specific requirements, both functional and non-functional ones.

Functional Requirements

The domain contributes to the high-level functional requirements in terms of facilitating the synchronization of changes. As the domain can only see encrypted transactions, refer to transaction privacy in the non-functional requirements, the functional requirements are satisfied on a lower level than the Daml transaction level.

Synchronization: The domain must facilitate the synchronization of the shared ledger among participants by establishing a total-order of transactions.
Transparency: The domain must inform the designated participants timely on changes to the shared ledger.
Finality: The domain must facilitate the synchronization of the shared ledger in an append-only fashion.
No unnecessary rejections: The domain should minimize unnecessary rejections of valid transactions.
Seek support for notifications: The domain must facilitate offset-based access to the notifications of the shared ledger.

Non-Functional Requirements

Reliability

Seamless fail-over for domain entities: All domain entities must be able to tolerate crash faults up to a certain failure rate, e.g., 1 sequencer node out of 3 can fail without interruption.
Resilience to faulty domain behavior: The domain must be able to detect and recover from failures of the domain entities, such as performing a fail-over on crash failures or retrying operations on transient failures if possible. The domain should tolerate byzantine failures of the domain entities.
Backups: The state of the domain entities have to be backed up such that in case of disaster recovery only minimal amount of data is lost.
Site-wide disaster recovery: In case of a failure of a data-center hosting a domain, the system must be able to fail-over to another data-center and recover operations.
Resilience to erroneous behavior: The domain must be resilient to erroneous behavior from the participants interacting with the domain.

Scalability

Horizontal scalability: The parallelizable domain entities and their sub-components must be able to horizontally scale.
Large transaction support: The domain entities must be able to cope with large transactions and their resulting large payloads.

Security

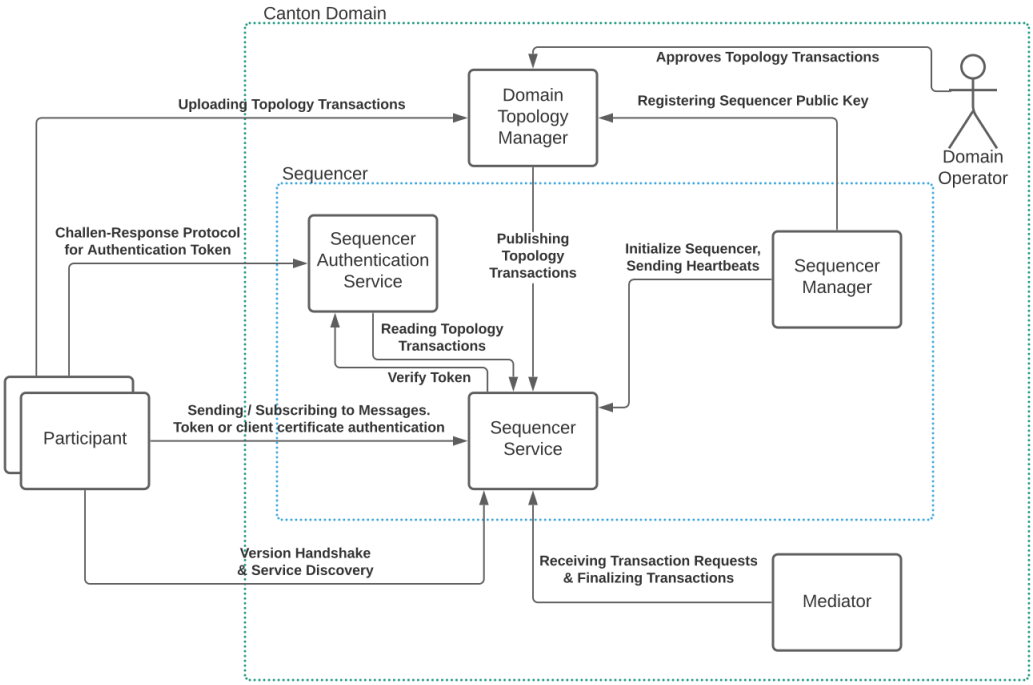
Domain entity compromise recovery: In case of a compromise of a domain entity, the domain must provide procedures to mitigate the impact of the compromise and allow to restore operations.
Standards compliant cryptography: All used cryptographic primitives and their configurations must comply to approved standards and based on existing and audited implementations.
Authentication and authorization: The participants interacting with the domain as well as the domain entities internal to the domain must authenticate themselves and have their appropriate permissions enforced.
Secure channel (TLS): All communication channels between the participants and the domain as well as between the domain entities themselves have to support a secure channel option using TLS, optionally with client certificate-based mutual authentication.
Distributed Trust: The domain should be able to be operated by a consortium in order to distribute the trust by the participants in the domain among many organizations.
Transaction Metadata Privacy: The domain entities must never learn the content of the transactions. The domain entities should learn a limited amount of transaction metadata, such as structural properties of a transaction and involved stakeholders.

Manageability

Garbage collection: The domain entities must provide ways to minimize the amount of data kept on hot storage, in particular data that is only required for auditability can move to cold storage or data that has been processed and stored by the participants could be removed after a specific retention period.
Upgradeability: The domain as a whole or individual domain entities must be able to upgrade with minimal downtime.
Semantic versioning: The interfaces, protocols, and persistent data schemas of the domain entities must be versioned according to semantic versioning guidelines.
Domain approved protocol versions: The domain must offer and verify the supported versions towards the participants. The domain must further ensure that the domain entities operate on compatible versions.
Reuse off-the-shelf solutions: The domain entities should use off-the-shelf solutions for persistence, API specification, logging, and metrics.
Metrics on communication and processing: The domain entities must expose metrics on communication and processing to facilitate operations and trouble shooting.
Component health monitoring: The domain entities must expose a health endpoint for monitoring.

Domain-Internal Components

The following diagram shows the architecture and components of a Canton domain as well as how a participant node interacts with the domain.



The domain consists of the following components:

Domain Service: The first point of contact for a participant node when connecting to a domain. The participant performs a version handshake with the domain service and discovers the available other services, such as the sequencer. If the domain requires a service agreement to be accepted by connecting participants, the domain service will provide the agreement.

Domain Topology Service: The domain topology services is responsible for all topology management operations on a domain. The service provides the essential topology state to a new participant node, i.e., the set of keys for the domain entities to bootstrap the participant node. Furthermore, participant nodes can upload their own topology transactions to the domain topology service, which inspects and possibly approves and publishes those topology transactions on the domain via the sequencer.

Sequencer Authentication Service: A node can authenticate itself to the sequencer service either using a client certificate or using an authentication token. The sequencer authentication service issues such authentication tokens after performing a challenge-response protocol with the node. The node has to sign the challenge with their private key corresponding to a public key that has been approved and published by the domain identity service.

Sequencer Service: The sequencer services establishes the total-order of messages, including transactions, within a domain. The service implements a total-order multicast, i.e., the sender of a message indicates the set of recipients to which the message is delivered. The order is established based on a unique timestamp assigned by the sequencer to each message.

Sequencer Manager: The sequencer manager is responsible for initializing the sequencer service.

Mediator: The mediator participates in the Canton transaction protocol and acts as the transaction commit coordinator to register new transaction requests and finalizes those requests by collecting transaction confirmations. The mediator provides privacy among the set of trans-

action stakeholders as the stakeholders do not communicate directly but always via the mediator.

The domain operator is responsible to operate the domain infrastructure and (optionally) also verifies and approves topology transactions, in particular to admit new participant nodes to a domain. The operator can either be a single entity managing the entire domain or a consortium of operators, refer to the distributed trust security requirement.

Drivers

Based on the set of domain internal components, a driver implements one or more components based on a particular technology. The prime component is the sequencer service and its ordering functionality, with implementations ranging from a relational database to a distributed blockchain. Components can be shared among integrations, for example, a mediator implemented on a relational database can be used together with a blockchain-based sequencer.

Canton Domain on Ethereum

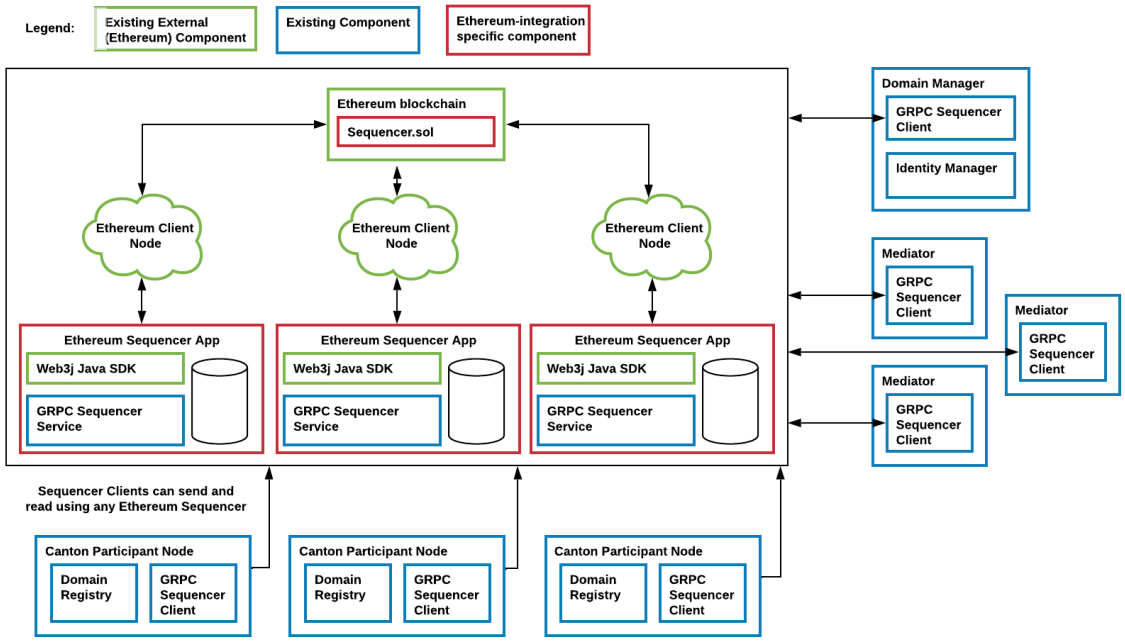
A Canton Ethereum domain uses a sequencer backed by Ethereum instead of by another ledger (such as Postgres or Fabric). The other domain components (mediator, domain manager) are reused from the relational database driver. Architecturally, the Canton Ethereum sequencer is a JVM application that interacts with an Ethereum client via the RPC JSON API to write events to the blockchain. Specifically, it interacts with an instance of the smart contract `Sequencer.sol` and calls function of `Sequencer.sol` to persist transactions and requests to the blockchain. It uses the configured Ethereum account to execute these calls. Analogous to the database-based sequencer implementations, multiple Ethereum sequencer applications can read and write to the same `Sequencer.sol` smart contract instance and they can do so through different Ethereum client nodes for high availability, scalability, and trust. The following diagrams shows the architecture of an Ethereum-based domain:

Note: When running in a multi-writer setup, each Ethereum Sequencer application needs to use a separate Ethereum account. Otherwise, transactions may get stuck due to nonce mismatches.

Smart contract Sequencer.sol

The smart contract deployed to the blockchain is implemented in Solidity. Its latest revision `drivers/ethereum/solidity/Sequencer.sol` is available in the enterprise edition only.

Data is written to the blockchain by emitting events to the transaction logs. The Sequencer Application reads all transactions (and transaction logs) created from calls to `Sequencer.sol` and keeps its own store for a view of the sequencer history. This enables the Sequencer Application to serve read subscriptions promptly without having to query the Ethereum client and to restart without having to re-read all the history. The store can either use in-memory storage or persistent storage (using a database).



Canton Domain on Fabric

Introduction to Hyperledger Fabric

Hyperledger Fabric is an open source enterprise-grade permissioned distributed ledger technology (DLT) platform.

Components of the Fabric Blockchain Network

The following key concepts of Fabric are relevant for the Canton domain integration with Fabric. For further details, refer to the Fabric documentation.

Peers: A network entity that maintains a Fabric ledger and runs chaincode containers in order to perform read/write operations to the Fabric ledger. Peers are owned and maintained by organizations.

Channels: A channel is a private blockchain overlay which allows for data isolation and confidentiality. A channel-specific Fabric ledger is shared across the peers in the channel, and transacting parties must be authenticated to a channel in order to interact with it. Members who are not a part of the channel are unable see the transactions or even know that the channel exists.

Ordering Service: Also known as orderer. A defined collective of nodes that orders transactions into a block and then distributes blocks to connected peers for validation and commit. The ordering service exists independent of the peer processes and orders transactions on a first-come-first-serve basis for all channels on the network.

Chaincode: A smart contract is code – invoked by a client application external to the blockchain network – that manages access and modifications to the current Fabric ledger state via transactions. In Hyperledger Fabric, smart contracts are packaged as chaincode. Chaincode is installed on peers and then defined and used on one or more channels. An endorsement policy specifies for each instantiation of a chaincode which peers have to validate and endorse a

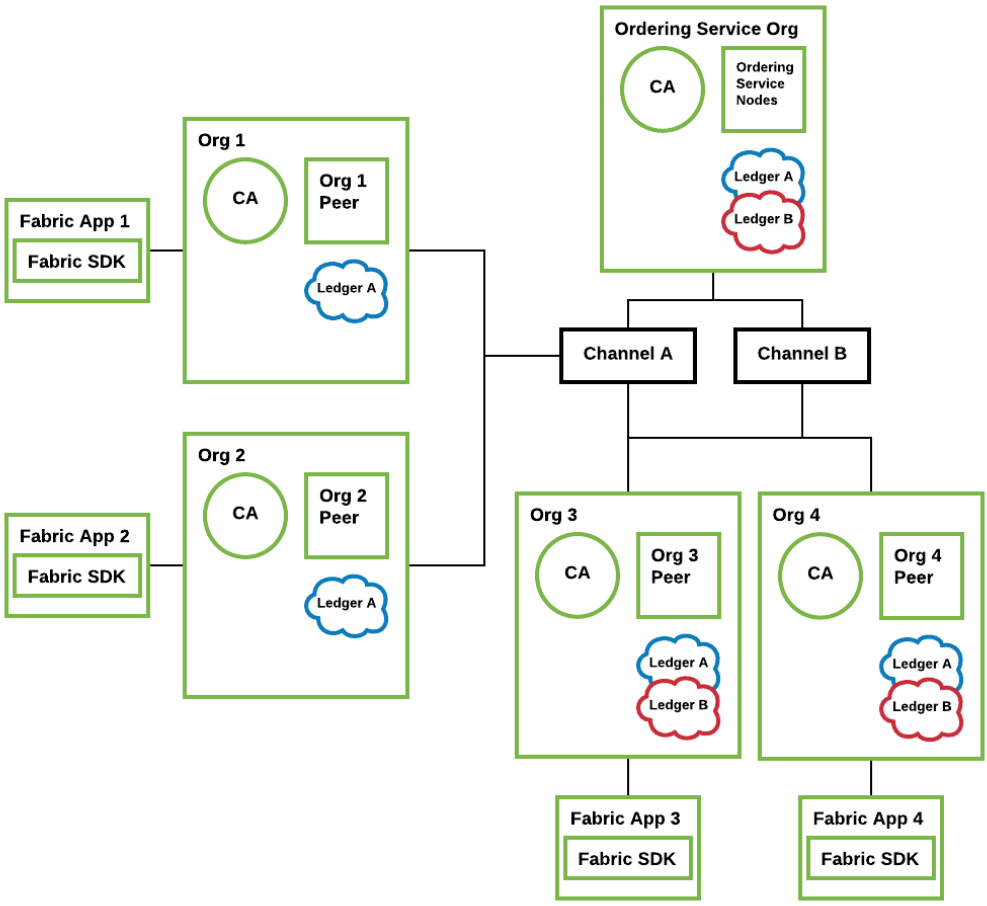
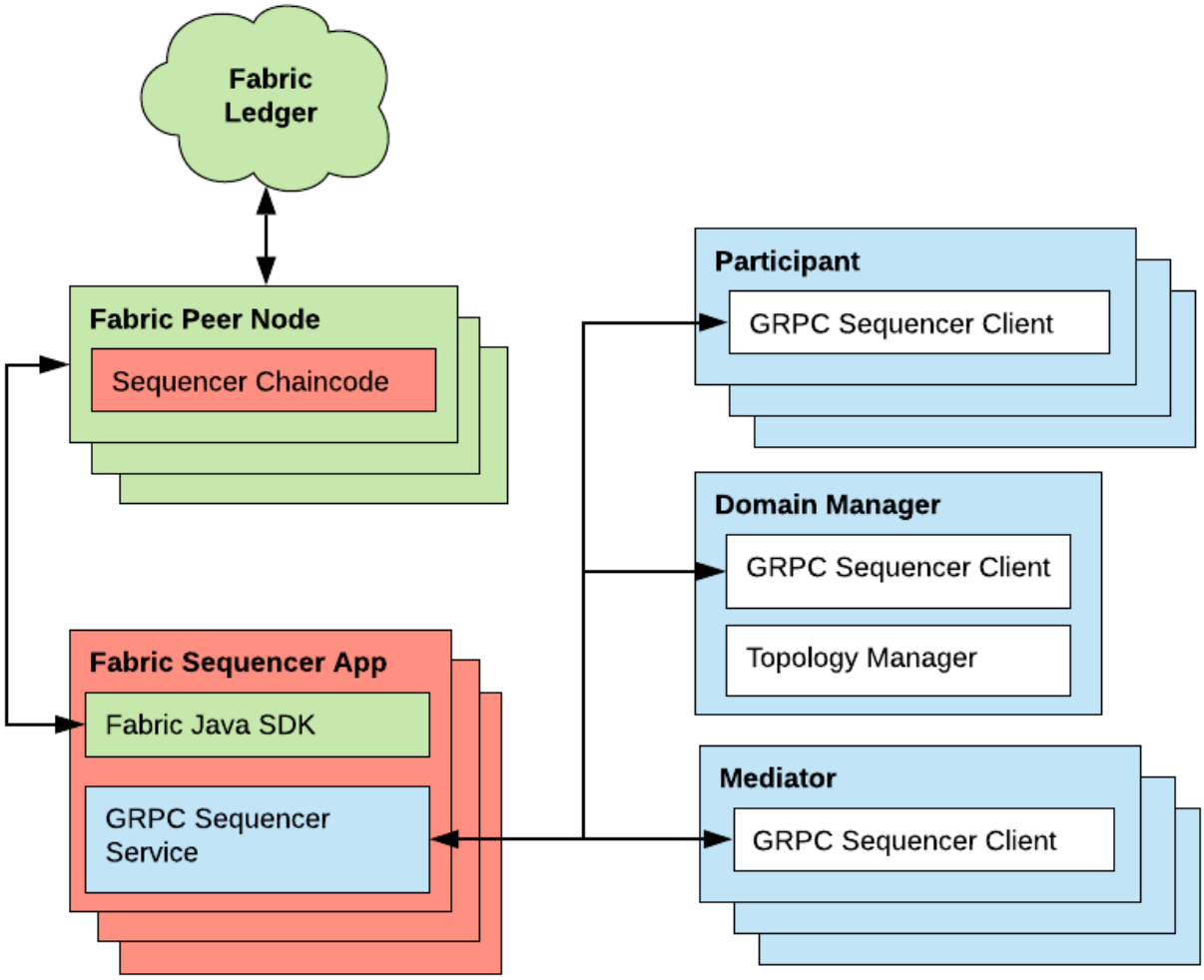


Fig. 32: An example Fabric blockchain network with four organizations. The ordering service has ordering nodes for ordering and distributing blocks on each of the channels defined under the ordering service. Channel A includes all four organizations, while channel B includes only Org 3 and Org 4. Authenticated client applications can send calls to their associated peers on the network.

transaction, such that the transaction is considered valid and part of the Fabric ledger.
Applications: Client applications in a Fabric-based network interact with the Fabric ledger using one of the available Fabric SDKs. Applications are able to propose changes to the ledger as well as to query the state of the ledger by using an identity issued by the organization's certificate authority (CA).

Architecture

In the v1 architecture of the Fabric driver, only the sequencer is integrated on top of Fabric. The other domain components are reused from the relational database driver. The Fabric-based sequencer supports running in a multi-writer, multi-reader topology for high availability, scalability, and trust. The following diagrams shows the architecture of a Fabric-based domain integration.



Fabric-based Sequencer

The Fabric Sequencer Application serves as an external standalone sequencer application that participants and other domain entities in a Canton network connect to in order to exchange ordered messages. It is an application that runs over Fabric by a consortium of organizations.

Typically each app operates via one Fabric client that belongs to a specific organization. These Fabric peers have visibility of the sequencer messages' metadata (sender and recipients of the messages), however the messages' payloads are fully encrypted.

A Canton domain requires beside the Sequencers one Domain Manager and one or more independently operated Mediators. All these nodes exclusively communicate with Participants via the Sequencer.

Participants trust the app they connect to and they can specify which one to connect to among the available ones. Participants could verify that Sequencer Applications are reporting consistent information by connecting to many or periodically checking other apps as they all need to report the same data.

The application supports a multi-writer, multi-reader architecture, such that multiple Fabric applications can operate on top of the same Fabric ledger. Sequencer clients within the Participants, Domain Manager or Mediators will communicate with the Sequencer Fabric Application and they can read or write from any of the available sequencer apps as they will have shared view of the Sequencer history for the domain.

Additionally, the same Fabric setup with a different channel can be used to operate different domains on the same Fabric infrastructure, since each channel contains a separate isolated Fabric ledger.

Sequencer Chaincode

The chaincode is implemented in Go. It supports:

Registering new members with the sequencer
Sending messages over the sequencer
- the messages are ordered by the Fabric ordering service and we subsequently use that order to define counters and timestamps
- if instead the order were defined in chaincode by keeping track of the last message counter, congestion would be created because the application would either have to process one message at a time or create a mechanism of batching messages to be processed in one transaction

The Sequencer Application reads all transactions created from chaincode operations and keeps its own store for a view of the sequencer history enabling them to serve read subscriptions promptly without having to constantly query chaincode and to restart without having to re-read all the history.

Analysis and Limitations

Below is an analysis with regard to driver requirements (functional and non-functional).

Functional Requirements

The Fabric driver must satisfy the following functional requirements:

Synchronization Fabric's ordering service establishes a total-order of transactions within a channel. A Canton domain is based on a single channel.

Transparency The Fabric blockchain ensures that all sequencer nodes obtain the same set of messages in the same order as established by the ordering service. The sequencer nodes inform their connected clients about their designated messages where the client is a recipient on.

Finality Fabric's ordering service provides finality, i.e., there will be no ledger forks and validated transactions will never be reverted.

Seek support for notifications The Fabric blockchain retains all sent messages and notifications. For efficiency purposes, the sequencer node caches the messages to satisfy read operations for a given offset without fetching the corresponding block.

Performance

The current performance we observe with the Fabric integration is around 15 tps of throughput and average latency of 800ms. Those numbers are based on local performance tests using the Daml Ledger API test tool with a simple 2 organizations with 1 peer each and 1 orderer node topology and a 2 of 2 endorsement policy.

Some factors that positively contribute to the current performance are:

Using Java for the SDK and Go for chaincode are good choices as opposed to something like Javascript for being compiled languages
We added more memory (2GB) to each peer and orderer node in our setup, which showed considerable performance improvement
The simplicity of the setup (only 2 peers, one orderer and all local)
Transactions are usually very small
Chaincode implementation is very simple
Some experiments were conducted with block cutting parameters such as max message count (max number of transactions that can exist in a block before a new block is cut) and batch timeout (max amount of time to wait before creating a block) in order to find a good balance of throughput and latency for our applications. A good tradeoff was found at 50 for max message count and 200ms for batch timeout, with an improvement for throughput at a slight increase in latency.
We are using LevelDB (instead of CouchDB).
We are using a round-robin load balancer when connecting to multiple sequencers and using the both sequencers' health and connectivity as a failover criteria.

This paper by IBM Research, India and this article by IBM discuss the many factors that can influence performance. This blog post also shares some Fabric performance best practices.

Reliability

Seamless fail-over for domain entities The sequencer can be deployed in a multi-writer and multi-reader topology (i.e. multiple sequencer nodes for the same domain) to achieve high availability. Since all Fabric sequencer nodes run on top of the same Fabric ledger, they will all see the same data and does not matter which sequencer is being used to write to and read from.

Additionally the Fabric sequencer node is backed by a database that caches the data read from the Fabric ledger such that in case of a crash it won't have to read the whole blockchain again. Instead it just needs to start reading the blocks from where it has last processed. The app also supports crash recovery.

On the client side, round-robin load balancing is used such that if one of the sequencer nodes goes down or becomes unhealthy clients will not route any requests to this sequencer. The sequencer provides a health endpoint that is used by clients for this purpose. It will indicate that it is unhealthy if it loses connection to the Fabric ledger or to its database.

Both the mediator and domain manager are also highly available via an active / passive mechanism (one active instance and 1-N passive replicas).

Resilience to faulty domain behavior Although Fabric supports for pluggable consensus protocols such as crash fault-tolerant (CFT) or byzantine fault tolerant (BFT) protocols that enable the platform to be customized to fit particular use cases and trust models, at the moment Fabric only offers a CFT ordering service implementation based on the Raft protocol.

Backups The backup procedures of the Fabric ledger must be used. The state of the sequencer node is just a cache and can be rehydrated from the state of the ledger.

Site-wide disaster recovery In a multi-writer, multi-reader topology, the sequencer nodes can be hosted by different organizations and across multiple datacenters to recover from the failure of an entire datacenter.

Resilience to erroneous behavior The Fabric sequencer node offers some resilience against an erroneous participant. For example, it checks that a client does not send messages to invalid recipients and only allows registered and authenticated clients to send messages. Clients are also required to sign their messages so sequencers can verify their origin, which prevents malicious sequencers from creating fake messages on behalf of specific members.

Scalability

Horizontal scalability Adding an additional sequencer to a domain is simply a matter of creating a new Fabric user and a new sequencer application with that configuration. A new Fabric organization and more Fabric peers could also be created, but this is optional. The setup will horizontally scale as well as a Fabric ledger will, which means performance could suffer if the Fabric topology is made more complex by adding peers and orderer nodes, in particular if their latency to each other is high. But there are ways to make up for that such as using a simpler endorsement policy that does not include all organizations in the setup. That's a trade-off between performance and trust that needs to be defined by the consortium.

Large transaction support Some Fabric platforms have a limit on the size of the block (commonly 99MB). This is therefore a hard limit that this sequencer has on the size of the transactions.

Security

Domain entity compromise recovery Without BFT support, a compromised orderer node cannot be recovered from automatically. Operational procedures, such as revoking the node's certificate, can limit further impact. Additionally, compromised peer nodes could endorse invalid transactions, but it would take a number of compromised peers enough to satisfy the endorsement policy to create incorrectly endorsed transactions on the ledger. All sequencer nodes must provide the same stream of messages, thus a compromised and malicious sequencer node can be detected if their stream differs.

Standards compliant cryptography The sequencer node and the other Canton domain entities use standard modern cryptography (EC-DSA with NIST curves and Ed25519 for signatures, AES128 GCM for symmetric encryption, SHA256 for hashes) provided by Tink/BouncyCastle. Fabric nodes can be deployed using cryptography provided by an HSM.

Authentication and authorization Authentication is implemented such that any sequencer client needs to be registered by the topology manager before they can connect. There are also authorization checks such as making sure that the declared sender is the currently authenticated client. And based on the type of member that is authenticated there are certain operations which may or may not be allowed.

Secure channel (TLS) The sequencer node provides an API secured with TLS. The Fabric network should be deployed according to its operations guide with TLS.

Distributed Trust A Fabric network can be operated by multiple organizations forming a consortium and distributing the trust among the organizations. The Mediator(s) and Domain Manager can only be operated by a single entity, so there is no distribution of trust for these nodes.

Transaction Metadata Privacy The sequencer node and the Fabric nodes (peers, orderer) learn the metadata of the transaction, in particular the stakeholders involved in the transaction.

Manageability

Garbage collection As Fabric is based on an immutable block-chain, processed sequencer messages cannot be removed. However there is a preview feature that allow messages to be removed by storing them in private data collections (which can be purged).

Upgradeability Upgrades of individual domain entities with minimal downtime not yet implemented.

Semantic versioning Canton is released under semantic versioning. The sequencer gRPC API is versioned with a major version number.

Domain approved protocol versions The authentication protocol validates the version compatibility between the sequencer nodes and the connecting node.

Reuse off-the-shelf solutions The local state of the sequencer node is stored in a relational database (Postgres).

Metrics on communication and processing Metrics are not yet fully implemented.

Component health monitoring The sequencer node contains basic health monitoring as an admin command.

1.44.2.3 Identity Management

Identity Providing Service

Every synchronization domain requires a shared and synchronized knowledge of identities and their associated keys among all participants and domain entities as the synchronisation protocol is built with the principle that provided the same data, all validators must come verifiably to the same result.

The service that establishes this shared understanding in a domain is the *Identity Providing Service* (**IPS**). From a synchronisation protocol perspective, the IPS is an abstract component and the synchronisation protocol only ever interacts with the read API of the IPS. There is no assumption on how the IPS is implemented, only the data it provides is relevant from a synchronisation perspective.

The participant nodes, the sequencer and the mediator have a local component called the *Identity Providing Service Client* (**IPS client**). This component establishes the connection to the IPS of the domain to read and validate the identity information in the domain.

The IPS client exposes a read API providing aggregated access to the domain topology information and public keys provided by the IPS of one or more domains.

The identity providing service receives keys and certificates through some process and evaluates the justifications, before presenting the information to the IPS clients of the participant or domain entities. The IPS clients verify the information. The local consumers of the IPS client read API trust the provided information without verifying the justifications, leading to a separation of synchronisation and identity management.

Requirements

The identity providing service describes the interface between the identity management process and the synchronisation functionality. It satisfies the high-level platform requirement on *identity information updates*. The following requirements are written from the perspective of the IPS client, i.e., the synchronisation layer components.

Mapping of Parties to Participants. I can query the state at a certain time and subscribe to a stream of updates associating a known identifier of a party to a set of participants as well as the local participant to a set of hosted parties. Mapping to a set of participants satisfies the high-level requirement on *parties using multiple participants*.
Participant Qualification. I can query the state at a certain time and subscribe to a stream of updates informing me about the trust level of a participant indicating either *untrusted* (trust level of 0) or *trusted* (trust level of 1).
Participant Relationship Qualification. A party to participant relationship is qualified, restricted to submission (including confirmation), confirmation, observation (read-only). This also satisfies the high-level requirement on *read-only participants*.
Domain aware mapping of Participants to Keys. I can query the state at a certain time and subscribe to a stream of updates mapping participants to a set of keys per synchronization domain.
Domain Entity Keys. I can query the current state and subscribe to a stream of updates on the keys of the domain entities.
Lifetime and Purpose of Keys. I can learn for any key that I receive for what it can be used, what cryptographic protocol it refers to and when it expires.

Signature Checking. Given a blob, a key I obtained from the IPS and a signature, I can verify that the signature is a valid signature for the given blob, signed with the respective key at a

certain time.
Immutability. The history of all keys is preserved within the same time boundaries as my audit logs such that I can always audit my participant or domain entity logs.
Evidence. For any data which I receive from the IPS I can get the set of associated evidence such that I can prove my arguments in a legal dispute. The associated evidence contains a descriptor which I can use to read up in the documentation on the definition of the otherwise opaque blob.
Race Condition Free. I can be sure that I am always certain about the validity of a key with respect to a transaction such that there cannot be a disagreement on the validity of a transaction due to an in-flight key change.
Querying for Parties. I can query, using an opaque query statement, the IPS for a party and will receive results based on a privacy policy not known to me.
Party metadata. I can access metadata associated with a party for display purposes.
Equivalent Trust Assumptions A federation protocol of the reference identity management service needs to be based on equivalent trust assumptions as the interoperability protocol such that there is no mismatch between the capabilities of the two.

Associated requirements that extend beyond the scope of the IPS:

API Versioning. I can use a versioned API which supports further extensions, see our general principles of upgradability and Software Versioning.
GDPR compliance. The identity providing service needs to comply with regulatory requirements such as the GRPR right to be forgotten.
Composability. The identity providing service needs to be composable such that I can add my own identity providing service based on the documentation and released binary artefacts.

Identity Management Design

While the previous section introduced the IPS as an abstract concept, we describe here the concrete implementation of our globally composable topology management system which incorporates identity. The design is introduced by first calling out a few basic design principles. We then introduce a formalism for the necessary topology management transactions. Finally, we connect the formalism to actual processes and cryptographic artefacts that describe the concrete implementation.

Design Principles

In order to understand the approach, a few key principles need to be introduced.

The synchronisation protocol is separated from the topology protocol. However, in order to leverage the composability properties of the synchronisation protocol, an equivalent approach is required for topology transactions. As such, given that there is no single globally trusted entity we can rely on for synchronisation, we also can't rely on a single globally trusted entity to establish identities, which leads us to the first principle:

Principle 1: For global synchronization to work in reality, there cannot be a single trust anchor.

A cryptographic key pair can uniquely be identified through the fingerprint of the public key. By owning the associated private key, an entity can always prove unambigously through a signature that the entity is the owner of the public key. We are using this principle heavily in our system to verify and authorize the activities of the participants. As such, we can introduce the second principle:

> *Principle 2: A participant is someone who can authorize and whose authorizations can be verified (someone with a known key)*

In short, a participant is someone with a key or with a set of keys that are known to belong together. However, the above definition doesn't mean that we necessarily know who owns the key. Ownership is an abstract aspect of the real world and is not relevant for the synchronisation itself. Real world ownership is only relevant for the interpretation of the meaning of some shared data, but not of the data processing itself.

Therefore, we introduce the third principle:

> *Principle 3: We separate certification of system identities and legal identities (or separation of cryptographical identity and metadata)*

Using keys, we can build trust chains by having a key sign a certificate certifying some ownership or some fact to be associated with another key. However, at the root of such chains is always the root key. The root key itself is not certified and the legal ownership cannot be verified: we just need to believe it. As an example, if we look at our local certificate store on our device, then we just believe that a certain root is owned by a named certificate authority. And our believe is rooted in the trust into our operating system provider that they have included only legitimate keys.

As such, any link between legal identities to cryptographic keys through certificates is based on a believe that the entity controlling the root key is honest and ensured that everybody attached to the trust-root has been appropriately vetted. Therefore, we can only believe that legal identities are properly associated, but verifying it in the absolute sense is very difficult, especially impossible online.

Another relevant aspect is that identity requirements are asymmetrical properties. While large corporations want to be known by their name (BANK), individuals tend to be more closed and would rather like that their identity is only revealed if really necessary (GDPR, HIPAA, confidential information, bank secrecy). Also, by looking at a bearer bond for example, the owner has a much higher interest in the identity of the obligor than the obligor has in the owner. If the obligor turns out to be bad or fraud, the owner might loose all their money. In contrast, the obligor doesn't really care to whom they are paying back the bond, except for some regulatory reasons. Therefore, we conclude the fourth principle

> *Principle 4: Identities on the ledger are an asymmetric problem, where privacy and publicity needs to be carefully weighted on a case by case basis.*

Formalism for a Global Composeable Topology System

Definitions

In order to construct a global composable topology system that incorporates identity, we introduce a topology scheme leading to globally unique identifiers. This allows us to avoid federation, which would require cooperation between identity providers or consensus among all participants and would be difficult to integrate with the synchronisation protocol.

We use (p_k^x, s_k^x) to refer to a public/private key pair of some cryptographic scheme, where the super-script x will provide the context of the usage of the key and the sub-script k will be used to distinguish keys.

In the following, we use the **fingerprint** of a public key $I_k = \text{fingerprint}(p_k)$ in order to refer to a key-pair (p_k, s_k).

Based on this, we use I_k, resp. (p_k, s_k), as an identity root key pair in the following. There can be multiple thereof and we do not make any statement on who the owner of such a key is.

Now, we introduce a globally unique identifier as a tuple (X, I_k) , where I_k refers to the previously introduced fingerprint of an identity root key and X is in principle some abstract identifier such that we can verify equality. As such, $(X, I_k) = (Y, I_l)$ if $X = Y$ and $I_k = I_l$. The identifier is globally unique by definition: there cannot be a collision as we defined two identifiers to be equal by definition if they collide. As such, the identity key I_k spans a **namespace** and guarantees that the namespace is, by definition, collision free.

The unique identifier within the project is defined as

```
/** A namespace spanned by the fingerprint of a pub-key
  *
  * This is based on the assumption that the fingerprint is unique to the public-
↪key
  */
final case class Namespace(fingerprint: Fingerprint) extends PrettyPrinting {
  def unwrap: String = fingerprint.unwrap
  def toProtoPrimitive: String = fingerprint.toProtoPrimitive
  def toLengthLimitedString: String68 = fingerprint.toLengthLimitedString
  override def pretty: Pretty[Namespace] = prettyOfParam(_.fingerprint)
}

/** a unique identifier within a namespace
  * Based on the Ledger API PartyIds/LedgerStrings being limited to 255
↪characters, we allocate
  * - 64 + 4 characters to the namespace/fingerprint (essentially SHA256 with
↪extra bytes),
  * - 2 characters as delimiters, and
  * - the last 185 characters for the Identifier.
  */
final case class UniqueIdentifier(id: Identifier, namespace: Namespace) extends
↪PrettyPrinting {
```

We use the global unique identifier to identify participant nodes $N = (N, I_k)$, parties $P = (P, I_k)$ and domain entities $D = (D, I_k)$ (which means that X is short for (X, I_k)). For parties P and participant nodes N, we should use a sufficiently long random number for privacy reasons. For domains D, we use readable names.

Incremental Changes

The topology state is build from incremental changes, so called topology transactions $\{+/-; \omega\}_t^{[s_k]}$ where $+$ is the addition and $-$ the subsequent removal. The incremental changes are not commutative and are ordered by time. For a given operand ω we note that the only accepted sequences are $+$ or $+-$, but that $-+$ or $--$ or $++$ are not accepted. The t denotes the time when the change was effected, i.e. when it was sequenced by the identity providing service.

The $\{.\}^{[s_k]}$ denotes the list of keys that authorized the change by signing the topology transaction. The authorization rules (which keys $[s_k]$ need to sign a topology transaction $\{.\}$) depend on the command ω. Most but not all transactions require the signatures to be nested in some form. Generally, anything that is distributed by the identity providing service needs to be signed with its key s_D and therefore $\forall \{.\}^{[s_k]} : s_D = \text{tail}\,[s_k]$.

For the sake of brevity, we will omit the identity providing service signature using s_D in the following and assume that it is always added upon distribution together with the timestamp t.

Topology Transactions

We can distinguish three types of topology transactions: identity delegations, mapping updates and domain governance updates. In the following, we will establish what these transactions mean and what they do and what the authorization rules are.

Delegation

The general delegation transaction is represented as

$$\{+/-; (?, I_k) \Rightarrow p_l\}^{s_k}$$

where the $?$ is a place-holder for a specific permissioning level. The delegation transaction indicates that a certain set of operations on the namespace spanned by the root key pair I_k is delegated to the public key p_l. The delegation is not exclusive, which means that there can be multiple keys that have to right to sign a specific transaction on the specific namespace.

There are two types of delegations:

> **namespace delegations**: $\{+/-; (*; I_k) \Rightarrow p_l\}^{[\tilde{s}_k]}$ which delegates to p_l the right to do all topology transactions on that particular namespace. The signature of such a delegated key is then considered to be equivalent to the signature of the root key: $s_l \simeq s_k$. If such a namespace delegation is a *root delegation*, then the delegated key is as powerful as the root key. If the *root delegation* flag is set to false, then the key can do everything on that namespace, except of issuing $NamespaceDelegation$. Therefore, such a delegation with the root delegation flag set to false effectively represents an intermediate CA, whereas with true, it's an equivalent root key. This operation is particularly useful to support offline storage of root keys, but as we will see later, it is also used to roll keys.

```
final case class NamespaceDelegation(
    namespace: Namespace,
    target: SigningPublicKey,
    isRootDelegation: Boolean,
) extends TopologyStateUpdateMapping {
```

> **identifier delegations**: $\{+/-; (X; I_k) \Rightarrow p_l\}^{[\tilde{s}_k]}$ which delegates the right to assign mappings to a particular identifier (X, I_k). With this right, the key holder can assign a party to a participant or run the party as a participant by assigning a key to it. This effectively represents a certificate.

```
final case class IdentifierDelegation(identifier: UniqueIdentifier, target:
↪SigningPublicKey)
    extends TopologyStateUpdateMapping {
```

From an authorization rule perspective, these delegations can delegate permissions to other keys and can be used to verify whether a certain key is allowed to sign a topology transaction. Therefore, we use for now the notation $\tilde{s}_k^I$ to indicate that some operation requires a signature of the root key s_k^I or by a key which was directly or indirectly authorized by the root key.

Mapping Updates

The generic second type of topology transactions are mapping updates which are represented as

$$\{+/-, (X, I_k) \rightarrow (Y, I_l, ct)\}^{[\tilde{s}_k, \tilde{s}_l]}$$

The above transaction maps one item of one namespace to something of a second namespace. For some mapping updates, the second namespace is always equal to the first namespace and we only require a single signature. The ct provides context to the mapping update and might include usage restrictions, depending on the type of mapping.

For transactions that require two signatures we support the composition of the add operation through

$$\{+, (X, I_k) \rightarrow (Y, I_l, ct)\}^{[\tilde{s}_k, \tilde{s}_l]} = \{+, (X, I_k) \rightarrow (Y, I_l, ct)\}^{[\tilde{s}_k]} + \{+, (X, I_k) \rightarrow (Y, I_l, ct)\}^{[\tilde{s}_l]}$$

and the removal operation through

$$\{-, (X, I_k) \rightarrow (Y, I_l, ct)\}^{[\tilde{s}_k, \tilde{s}_l]} = \{-, (X, I_k) \rightarrow (Y, I_l, ct)\}^{[\tilde{s}_k]} || \{-, (X, I_k) \rightarrow (Y, I_l, ct)\}^{[\tilde{s}_l]}$$

There are four different sub-types of valid mapping transactions:

Domain Keys: The mapping transaction of $\{+, D \rightarrow (p_D, \text{ct})\}^{s_D}$ updates the keys for the domain entities. Valid qualifiers for ct are *identity*, *sequencer*, *mediator*. As every state update needs to be signed by the domain, the domain definition corresponds to the initial seed of the identity transaction stream $\{D \rightarrow (p_D, \text{identity})\}^{s_D}$. If a participant knows the domain id of D, it can verify that this initial seed is correctly authorized by the owner of the key governing the unique identifier of the domain id.

Owner to Key Mappings: The mapping transaction $\{+, (N, I_k) \rightarrow (p_l, \text{ct})\}^{[\tilde{s}_k]}$ updates the keys that are associated with an owner such as a participant or a domain entity. The key purposes can be *signing* and/or *encryption*. If more than one key is defined, all systems are supposed to use the key that was observed first and is still active.

```
final case class OwnerToKeyMapping(owner: KeyOwner, key: PublicKey)
    extends TopologyStateUpdateMapping {
```

Party to Participant Mappings: The mapping transaction $\{+, (P, I_k) \rightarrow (N, I_l, \text{ct})\}^{[\tilde{s}_k, \tilde{s}_l]}$ maps a party to a participant. The context ct would call out the permissions such as *submission*, *confirmation* or *observation*.

```
final case class PartyToParticipant(
    side: RequestSide,
    party: PartyId,
    participant: ParticipantId,
    permission: ParticipantPermission,
) extends TopologyStateUpdateMapping {
```

Participant State Updates

The fourth type of topology transactions are participant state updates as domain governance transactions $\{d|a|c|p, N\}^{s_D}$. Here, d means *disabled* (participant cannot be involved in any transaction), a means participant is *active*, c means participant cannot submit transactions but only *confirm*, p means participant is *purged* and will never be back again. Participant states are owned by the operator of the committer. It is at the committers discretion to decide whether a participant is allowed to use the domain or not.

```
final case class ParticipantState(
    side: RequestSide,
    domain: DomainId,
    participant: ParticipantId,
    permission: ParticipantPermission,
    trustLevel: TrustLevel,
) extends TopologyStateUpdateMapping {

  require(
    permission.canConfirm || trustLevel == TrustLevel.Ordinary,
    "participant trust level must either be ordinary or permission must be
↪confirming",
  )
```

Some Considerations

Removal Authorizations

We note that the authorization rules for the addition are more strict than for the removal: Any removal can be authorized by the domain key s_D such that the domain operator can prune the topology state if necessary, which is fine, as the accessibility of a domain is anyway dependent on the cooperation of the domain operator.

Therefore, when talking about removal authorization, we explain the authorization check the IPS will make if it receives a removal request from an untrusted source. Consequently, all participants will at least be aware whether a certain topology transaction removal was authorized by the domain topology manager or by the actual authority of that topology transaction.

Revocations

One important point to note is that all topology transactions have a local effect. This means that a removal of a root key $\{-, p_k\}$ will not invalidate all transactions that have been signed before by the key directly or indirectly. Therefore, to revoke a key as in invalidating everything the key has signed requires publishing a set of topology transactions together.

Domain Topology State

Looking at the given formalism, we can distinguish between the *topology state* and the *domain topology state*. The difference between these two is that the *topology state* is comprised of all delegation and mapping transactions. The domain topology state extends this definition by adding *domain governance updates* such as participant states. And the domain topology state overrides the authorization rule by allowing a domain to remove any previous topology transaction.

Bootstrapping

Based on the above explanations, we observe that the authorized domain topology state is given by all signed and properly authorized topology transaction which additionally have been ordered and signed by the domain topology manager and distributed (and signed) by the sequencer. Consequently, for a new participant connecting to a domain, in order to validate the topology state and know that they are talking to the right sequencer, it only needs to know the unique-identifier of the domain. Using this unique identifier, it can verify the authenticity and correctness of the topology state, as it can verify the correct authorization of the corresponding topology transactions.

This is the bootstrapping problem of any Canton network: In order to connect to a domain, a participant needs to know the domain id (a unique identifier) of a domain, which it needs to receive through a trusted channel.

Default Party

Given that (N, I_k) and (P, I_k) are both unique identifiers which we use to refer to participants and parties, we can introduce for every participant its default party. This provides a more straight forward meaning of a party as being a virtualisation concept on top of the synchronisation structure.

Therefore, any party in the system can either self-host on a participant, or delegate the hosting to another participant. Or do a mixture of both.

Submission vs Confirmation

Due to sub-transaction privacy, validating participants only learn the identity of the submitter if they are stakeholders of the root transaction node. A malicious participant with *confirmation* permissions for a party can submit transactions in the name of the party. Such a behaviour will be detected by any other participant hosting the party, but these participants cannot prevent the transaction from being accepted. Therefore, the distinction between *submission* and *confirmation* permissions in the party to participant mappings are only guaranteed to be respected by the software provided by Digital Asset.

Topology State Accumulation

Now, we define the topology state S_t at time t as provided by the identity service provider of a domain incrementally as

$$\begin{aligned} S_t &= S_{t-1} + \{., \omega\}_t^{s_k, s_{k'}} \\ &= \bigoplus_{t'<t} \{., \omega\}_{t'}^{[s.]} \\ &= [(., I) \Rightarrow p] + [((X, I) \to Y)] + [(., N)] \end{aligned}$$

Here, the first expression on the last line represents the delegations, the second corresponds to the mapping updates and the third one to the participant state updates.

We assume that the identity providing service (which is part of the committer) is presented by someone with a topology transaction $\{.\}^{s_k}$. Upon a vetting operation where the operator can decide if the proposed change is acceptable, the IPS sequences, validates, signs (using the domain key s_D) and distributes the topology state changes to all affected domain entities.

Privacy by Design

A tricky question is how to provide privacy by design, allowing participants only to learn about other parties and participants on a need to know basis, while still ensuring that enough information is available for the participant to progress and ensuring that the information remains immutable and verifiable.

We do this by generally restricting what is shared with participants by default. Instead of broadcasting the mappings $X \to Y$ to all participants, we broadcast $T = (H(X), t_i d)$ instead.

We include a service with the committer that allows to query the data once the left hand side has been learnt. This means that once X of $H(X)$ is known, a participant can call a service that returns the corresponding topology transactions, which in turn can be verified to be justified.

Looking at the participant to key mappings $N \to K$ we note that by only broadcasting $H(N)$ instead of N, other participants cannot transact with a participant P unless they have learned P's identity. This is a similar property as we see with phone numbers. Guessing a phone number is hard. However, once we receive a call from a phone, we know the calling number.

By restricting the data we broadcast about the party to participant mappings, we prevent two aspects. First, nobody can contact a party unless they have learned the party identifier before. This is important as otherwise, any participant on the ledger might e.g. contact all parties of another participating bank. Second, we also protect that somebody can know how many parties e.g. a participant manages. This prevents learning questions such as how many parties are represented by a certain participant (how many clients does my competitor have).

Cross-Domain Delegations

In our design of participants and parties, we observe that a participant is a system entity whereas a party is meant to represent some actor in the real world. In order to commoditise the ledger as a service, we need to provide a way that makes a party something fluid that can be moved around from participant. As the participant should just be a service, it might be acceptable to keep it pinned to an identity domain. But a party should be able to travel but still be held accountable for the obligations.

Permissioning a party on a second participant node that exists in the same domain is already possible in the present formalism: $\{(P, I_k) \to (N_2, I_k)\}^{s_k}$

A straight-forward extension to permission a party on a second participant in another identity namespace is: $\{(P, I_k) \to (N_2, I_l)\}^{s_k, s_l}$ Based on the additivity of such statements, we can also build such a permission from two individually signed transactions.

The party delegation transaction supports delegating the permissioning of a party to a key outside of the root key namespace: $\{(P, I_k) \Rightarrow p_l\}^{s_k}$

Multi-Domain Transaction

The key challenge of the identity management aspect is to design it such that we can support multi-domain synchronisation without requiring the committers to cooperate.

First, we note that we avoid collision problems by design using globally unique identifiers derived from namespaces generated by root keys.

Second, we note that we do not need complete consistency of identities between the committers. All that is required is a sufficient overlap.

We first introduce a new mapping transaction denoting the transfer permission as $\{P \to D_T\}$ on the source domain D_S. The transfer permission means that for the given party, out-transfers of contracts to the target domain D_T are allowed. However, this does not imply that the target domain has a corresponding permission to move the contract back. It might, but there is no guarantee.

Right now, in the transfer-out protocol, the transfer-out request check reads *The target domain is acceptable to all stakeholders*. By introducing $\{P \to D\}$ we are now explicit about what an acceptable domain is: for all stakeholder parties of the particular contract, there is an approriate transfer permission on the current domain.

However, there are edge cases we need to deal with: what happens if on domain D_T, the party P doesn't exist? Or what happens if the participants representing P on D_S are completely different than on D_T? This can happen either due to a misconfiguration or due to a race-condition of an inflight change.

Clearly, in such a case, the transfer must fail in a predicatable manner. Therefore, we introduce two new rules

1) transfer-out on D_S will be rejected if $(P \to [N])^{t_1}_{D_S} \cap (P \to [N])^{t_0}_{D_T} = \emptyset$

2) transfer-in on D_T will be rejected if $(P \to [N])^{t_1}_{D_S} \cap (P \to [N])^{t_2}_{D_T} = \emptyset$

These rules boil down to the simple verbal requirement that at least one participant representing the affected party needs to be present on both domains while the transfer takes place from t_0 to t_2.

Validation

Scenario: *How to roll participant keys?*

This corresponds to $\{+,(N,I_k) \rightarrow p_2\}^{\tilde{s}_k}\{-,(N,I_k) \rightarrow p_1\}^{\tilde{s}_k}$

Scenario: *I can setup my local committer and my local participant and subsequently connect to a remote committer.*

Either locally create an identity key and get it vetted by the committer. Or get $IdentifierDelegations$ by another identity key holder, load it locally into the identity store, subsequently pushing to a remote committer.

Scenario: *I can register a party on multiple participants?*

$\{+,P \rightarrow N_1\}\{+,\rightarrow N_2\}$

Scenario: *I can introduce a new cryptographic signing scheme without loosing my identities or I can roll a root identity key.*

Assuming that $\{I_k^S\}$ is the original key of scheme S and we want to use scheme S', then the following transaction should suffice: $\{J_k^{S'}\}^{I_k^S}$. Now the new key is endorsed to act on the namespace originally spanned by I_k. If furthermore I_k is revoked, then the new key becomes the root key. If the signature of the old key is not trusted then the delegation needs to be believed .

There is a corresponding RFC for X509s for that https://tools.ietf.org/html/rfc6489

Scenario: *I can migrate a party from one participant to another.*

$$\{+,(P,I_k) \rightarrow (N_2,I_l)\}^{I_k,I_l}\{-,P \rightarrow (N_2,I_k)\}^{I_k}$$

Implementation

Domain Id

We assume that the domain id is shared with the connecting participant through a trusted channel. This can be implemented as a secure out of band process or by trusting TLS server authentication when initially requesting the domain id from the *Sequencer Service*.

Identity Providing Service API

The Identity Providing Service client API is defined as follows:

```
/** Client side API for the Identity Providing Service. This API is used to get
↪information about the layout of
  * the domains, such as party-participant relationships, used encryption and
↪signing keys,
  * package information, participant states, domain parameters, and so on.
  */
class IdentityProvidingServiceClient {

  private val domains = TrieMap.empty[DomainId, DomainTopologyClient]

  def add(domainClient: DomainTopologyClient): this.type = {
```

(continues on next page)

(continued from previous page)

```
    domains += (domainClient.domainId -> domainClient)
    this
  }

  def allDomains: Iterable[DomainTopologyClient] = domains.values

  def tryForDomain(domain: DomainId): DomainTopologyClient =
    domains.getOrElse(domain, sys.error("unknown domain " + domain.toString))

  def forDomain(domain: DomainId): Option[DomainTopologyClient] = domains.
↪get(domain)

}

trait TopologyClientApi[+T] { this: HasFutureSupervision =>

  /** The domain this client applies to */
  def domainId: DomainId

  /** Our current snapshot approximation
    *
    * As topology transactions are future dated (to prevent sequential
↪bottlenecks), we do
    * have to "guess" the current state, as time is defined by the sequencer after
    * we've sent the transaction. Therefore, this function will return the
    * best snapshot approximation known.
    *
    * The snapshot returned by this method should be used when preparing a
↪transaction or transfer request (Phase 1).
    * It must not be used when validating a request (Phase 2 - 7); instead, use
↪one of the `snapshot` methods with the request timestamp.
    */
  def currentSnapshotApproximation(implicit traceContext: TraceContext): T

  /** Possibly future dated head snapshot
    *
    * As we future date topology transactions, the head snapshot is our latest
↪knowledge of the topology state,
    * but as it can be still future dated, we need to be careful when actually
↪using it: the state might not
    * yet be active, as the topology transactions are future dated. Therefore, do
↪not act towards the sequencer
    * using this snapshot, but use the currentSnapshotApproximation instead.
    */
  def headSnapshot(implicit traceContext: TraceContext): T = checked(
    trySnapshot(topologyKnownUntilTimestamp)
  )

  /** The approximate timestamp
    *
    * This is either the last observed sequencer timestamp OR the effective
↪timestamp after we observed
    * the time difference of (effective - sequencer = epsilon) to elapse
    */
  def approximateTimestamp: CantonTimestamp
```

(continues on next page)

(continued from previous page)

```
  /** The most recently observed effective timestamp
    *
    * The effective timestamp is sequencer_time + epsilon(sequencer_time), where
    * epsilon is given by the topology change delay time, defined using the
↪domain parameters.
    *
    * This is the highest timestamp for which we can serve snapshots
    */
  def topologyKnownUntilTimestamp: CantonTimestamp

  /** Returns true if the topology information at the passed timestamp is already
↪known */
  def snapshotAvailable(timestamp: CantonTimestamp): Boolean

  /** Returns the topology information at a certain point in time
    *
    * Use this method if you are sure to be synchronized with the topology state
↪updates.
    * The method will block & wait for an update, but emit a warning if it is not
↪available
    *
    * The snapshot returned by this method should be used for validating
↪transaction and transfer requests (Phase 2 - 7).
    * Use the request timestamp as parameter for this method.
    * Do not use a response or result timestamp, because all validation steps
↪must use the same topology snapshot.
    */
  def snapshot(timestamp: CantonTimestamp)(implicit traceContext: TraceContext):
↪Future[T]
  def snapshotUS(timestamp: CantonTimestamp)(implicit
      traceContext: TraceContext
  ): FutureUnlessShutdown[T]

  /** Waits until a snapshot is available
    *
    * The snapshot returned by this method should be used for validating
↪transaction and transfer requests (Phase 2 - 7).
    * Use the request timestamp as parameter for this method.
    * Do not use a response or result timestamp, because all validation steps
↪must use the same topology snapshot.
    */
  def awaitSnapshot(timestamp: CantonTimestamp)(implicit traceContext:
↪TraceContext): Future[T]

  /** Supervised version of [[awaitSnapshot]] */
  def awaitSnapshotSupervised(description: => String, warnAfter: Duration = 30.
↪seconds)(
      timestamp: CantonTimestamp
  )(implicit
      traceContext: TraceContext
  ): Future[T] = supervised(description, warnAfter)(awaitSnapshot(timestamp))

  /** Shutdown safe version of await snapshot */
  def awaitSnapshotUS(timestamp: CantonTimestamp)(implicit
      traceContext: TraceContext
  ): FutureUnlessShutdown[T]
```

(continues on next page)

(continued from previous page)

```
  /** Supervised version of [[awaitSnapshotUS]] */
  def awaitSnapshotUSSupervised(description: => String, warnAfter: Duration = 30.
↪seconds)(
      timestamp: CantonTimestamp
  )(implicit
      traceContext: TraceContext
  ): FutureUnlessShutdown[T] = supervisedUS(description,
↪warnAfter)(awaitSnapshotUS(timestamp))

  /** Returns the topology information at a certain point in time
    *
    * Fails with an exception if the state is not yet known.
    *
    * The snapshot returned by this method should be used for validating
↪transaction and transfer requests (Phase 2 - 7).
    * Use the request timestamp as parameter for this method.
    * Do not use a response or result timestamp, because all validation steps
↪must use the same topology snapshot.
    */
  def trySnapshot(timestamp: CantonTimestamp)(implicit traceContext:
↪TraceContext): T

  /** Returns an optional future which will complete when the timestamp has been
↪observed
    *
    * If the timestamp is already observed, we return None.
    *
    * Note that this function allows to wait for effective time (true) and
↪sequenced time (false).
    * If we wait for effective time, we wait until the topology snapshot for that
↪given
    * point in time is known. As we future date topology transactions (to avoid
↪bottlenecks),
    * this might be before we actually observed a sequencing timestamp.
    */
  def awaitTimestamp(
      timestamp: CantonTimestamp,
      waitForEffectiveTime: Boolean,
  )(implicit traceContext: TraceContext): Option[Future[Unit]]

  def awaitTimestampUS(
      timestamp: CantonTimestamp,
      waitForEffectiveTime: Boolean,
  )(implicit traceContext: TraceContext): Option[FutureUnlessShutdown[Unit]]

}

/** The client that provides the topology information on a per domain basis
  */
trait DomainTopologyClient extends TopologyClientApi[TopologySnapshot] with
↪AutoCloseable {
  this: HasFutureSupervision =>

  /** Wait for a condition to become true according to the current snapshot
↪approximation
```

(continues on next page)

(continued from previous page)

```
    *
    * @return true if the condition became true, false if it timed out
    */
  def await(condition: TopologySnapshot => Future[Boolean], timeout:
↪Duration)(implicit
      traceContext: TraceContext
  ): FutureUnlessShutdown[Boolean]

}

trait BaseTopologySnapshotClient {

  protected implicit def executionContext: ExecutionContext

  /** The official timestamp corresponding to this snapshot */
  def timestamp: CantonTimestamp

  /** Internally used reference time (representing when the last change happened
↪that affected this snapshot) */
  def referenceTime: CantonTimestamp = timestamp

}

/** The subset of the topology client providing party to participant mapping
↪information */
trait PartyTopologySnapshotClient {

  this: BaseTopologySnapshotClient =>

  /** Load the set of active participants for the given parties */
  def activeParticipantsOfParties(
      parties: Seq[LfPartyId]
  ): Future[Map[LfPartyId, Set[ParticipantId]]]

  def activeParticipantsOfPartiesWithAttributes(
      parties: Seq[LfPartyId]
  ): Future[Map[LfPartyId, Map[ParticipantId, ParticipantAttributes]]]

  /** Returns the set of active participants the given party is represented by as
↪of the snapshot timestamp
    *
    * Should never return a PartyParticipantRelationship where
↪ParticipantPermission is DISABLED.
    */
  def activeParticipantsOf(
      party: LfPartyId
  ): Future[Map[ParticipantId, ParticipantAttributes]]

  /** Returns Right if all parties have at least an active participant passing
↪the check. Otherwise, all parties not passing are passed as Left */
  def allHaveActiveParticipants(
      parties: Set[LfPartyId],
      check: (ParticipantPermission => Boolean) = _.isActive,
  ): EitherT[Future, Set[LfPartyId], Unit]

  /** Returns the consortium thresholds (how many votes from different
↪participants that host the consortium party
```

(continues on next page)

(continued from previous page)

```
    * are required for the confirmation to become valid). For normal parties
↪returns 1.
    */
  def consortiumThresholds(parties: Set[LfPartyId]): Future[Map[LfPartyId,
↪PositiveInt]]

  /** Returns the Authority-Of delegations for consortium parties. Non-consortium
↪parties delegate to themselves
    * with threshold one
    */
  def authorityOf(parties: Set[LfPartyId]): Future[AuthorityOfResponse]

  /** Returns true if there is at least one participant that satisfies the
↪predicate */
  def isHostedByAtLeastOneParticipantF(
      party: LfPartyId,
      check: ParticipantAttributes => Boolean,
  ): Future[Boolean]

  /** Returns the participant permission for that particular participant (if
↪there is one) */
  def hostedOn(
      partyId: LfPartyId,
      participantId: ParticipantId,
  ): Future[Option[ParticipantAttributes]]

  /** Returns true of all given party ids are hosted on a certain participant */
  def allHostedOn(
      partyIds: Set[LfPartyId],
      participantId: ParticipantId,
      permissionCheck: ParticipantAttributes => Boolean = _.permission.isActive,
  ): Future[Boolean]

  /** Returns whether a participant can confirm on behalf of a party. */
  def canConfirm(
      participant: ParticipantId,
      party: LfPartyId,
      requiredTrustLevel: TrustLevel = TrustLevel.Ordinary,
  ): Future[Boolean]

  /** Returns all active participants of all the given parties. Returns a Left if
↪some of the parties don't have active
    * participants, in which case the parties with missing active participants
↪are returned. Note that it will return
    * an empty set as a Right when given an empty list of parties.
    */
  def activeParticipantsOfAll(
      parties: List[LfPartyId]
  ): EitherT[Future, Set[LfPartyId], Set[ParticipantId]]

  def partiesWithGroupAddressing(
      parties: Seq[LfPartyId]
  ): Future[Set[LfPartyId]]

  /** Returns a list of all known parties on this domain */
  def inspectKnownParties(
```

(continues on next page)

(continued from previous page)

```
      filterParty: String,
      filterParticipant: String,
      limit: Int,
  ): Future[
    Set[PartyId]
  ] // TODO(#11255): Decide on whether to standarize APIs on LfPartyId or PartyId
↪and unify interfaces

}

object PartyTopologySnapshotClient {
  final case class AuthorityOfDelegation(expected: Set[LfPartyId], threshold:
↪PositiveInt)

  def nonConsortiumPartyDelegation(partyId: LfPartyId): AuthorityOfDelegation =
    AuthorityOfDelegation(Set(partyId), PositiveInt.one)

  final case class AuthorityOfResponse(response: Map[LfPartyId,
↪AuthorityOfDelegation])

  final case class PartyInfo(
      groupAddressing: Boolean,
      threshold: PositiveInt, // > 1 for consortium parties
      participants: Map[ParticipantId, ParticipantAttributes],
  )

  object PartyInfo {
    def nonConsortiumPartyInfo(participants: Map[ParticipantId,
↪ParticipantAttributes]): PartyInfo =
      PartyInfo(groupAddressing = false, threshold = PositiveInt.one,
↪participants = participants)

    lazy val EmptyPartyInfo: PartyInfo = nonConsortiumPartyInfo(Map.empty)
  }
}

/** The subset of the topology client, providing signing and encryption key
↪information */
trait KeyTopologySnapshotClient {

  this: BaseTopologySnapshotClient =>

  /** returns newest signing public key */
  def signingKey(owner: KeyOwner): Future[Option[SigningPublicKey]]

  /** returns all signing keys */
  def signingKeys(owner: KeyOwner): Future[Seq[SigningPublicKey]]

  /** returns newest encryption public key */
  def encryptionKey(owner: KeyOwner): Future[Option[EncryptionPublicKey]]

  /** returns all encryption keys */
  def encryptionKeys(owner: KeyOwner): Future[Seq[EncryptionPublicKey]]

  /** Returns a list of all known parties on this domain */
  def inspectKeys(
```

(continues on next page)

(continued from previous page)

```
      filterOwner: String,
      filterOwnerType: Option[KeyOwnerCode],
      limit: Int,
  ): Future[Map[KeyOwner, KeyCollection]]

}

/** The subset of the topology client, providing participant state information */
trait ParticipantTopologySnapshotClient {

  this: BaseTopologySnapshotClient =>

  // used by domain to fetch all participants
  @Deprecated(since = "3.0")
  def participants(): Future[Seq[(ParticipantId, ParticipantPermission)]]

  /** Checks whether the provided participant exists and is active */
  def isParticipantActive(participantId: ParticipantId): Future[Boolean]

}

/** The subset of the topology client providing mediator state information */
trait MediatorDomainStateClient {
  this: BaseTopologySnapshotClient =>

  /** returns the list of currently known mediators */
  @deprecated(since = "2.7", message = "Use mediatorGroups instead.")
  final def mediators(): Future[Seq[MediatorId]] =
    mediatorGroups().map(_.flatMap(mg => mg.active ++ mg.passive))

  def mediatorGroups(): Future[Seq[MediatorGroup]]

  def isMediatorActive(mediatorId: MediatorId): Future[Boolean] =
    mediatorGroups().map(_.exists { group =>
      // Note: mediator in group.passive should still be able to authenticate and
↪process MediatorResponses,
      // only sending the verdicts is disabled and verdicts from a passive
↪mediator should not pass the checks
      group.isActive && (group.active.contains(mediatorId) || group.passive.
↪contains(mediatorId))
    })

  def isMediatorActive(mediator: MediatorRef): Future[Boolean] = {
    mediator match {
      case MediatorRef.Single(mediatorId) =>
        isMediatorActive(mediatorId)
      case MediatorRef.Group(mediatorsOfDomain) =>
        mediatorGroup(mediatorsOfDomain.group).map {
          case Some(group) => group.isActive
          case None => false
        }
    }
  }

  def mediatorGroupsOfAll(
      groups: Seq[MediatorGroupIndex]
```

(continues on next page)

(continued from previous page)

```
  ): EitherT[Future, Seq[MediatorGroupIndex], Seq[MediatorGroup]] =
    if (groups.isEmpty) EitherT.rightT(Seq.empty)
    else
      EitherT(
        mediatorGroups()
          .map { mediatorGroups =>
            val existingGroupIndices = mediatorGroups.map(_.index)
            val nonExisting = groups.filterNot(existingGroupIndices.contains)
            Either.cond(
              nonExisting.isEmpty,
              mediatorGroups.filter(g => groups.contains(g.index)),
              nonExisting,
            )
          }
      )

  def mediatorGroup(index: MediatorGroupIndex): Future[Option[MediatorGroup]] = {
    mediatorGroups().map(_.find(_.index == index))
  }
}

/** The subset of the topology client providing sequencer state information */
trait SequencerDomainStateClient {
  this: BaseTopologySnapshotClient =>

  /** returns the sequencer group */
  def sequencerGroup(): Future[Option[SequencerGroup]]
}

// this can be removed with 3.0
@Deprecated(since = "3.0")
trait CertificateSnapshotClient {

  this: BaseTopologySnapshotClient =>

  @Deprecated(since = "3.0.0")
  def hasParticipantCertificate(participantId: ParticipantId)(implicit
      traceContext: TraceContext
  ): Future[Boolean] =
    findParticipantCertificate(participantId).map(_.isDefined)

  @Deprecated(since = "3.0.0")
  def findParticipantCertificate(participantId: ParticipantId)(implicit
      traceContext: TraceContext
  ): Future[Option[X509Cert]]

}

trait VettedPackagesSnapshotClient {

  this: BaseTopologySnapshotClient =>

  /** Returns the set of packages that are not vetted by the given participant
    *
    * @param participantId the participant for which we want to check the package
↪vettings
```

(continues on next page)

(continued from previous page)

```
    * @param packages the set of packages that should be vetted
    * @return Right the set of unvetted packages (which is empty if all packages
↪are vetted)
    *         Left if a package is missing locally such that we can not verify
↪the vetting state of the package dependencies
    */
  def findUnvettedPackagesOrDependencies(
      participantId: ParticipantId,
      packages: Set[PackageId],
  ): EitherT[Future, PackageId, Set[PackageId]]

}

trait DomainGovernanceSnapshotClient {
  this: BaseTopologySnapshotClient with NamedLogging =>

  def findDynamicDomainParametersOrDefault(
      protocolVersion: ProtocolVersion,
      warnOnUsingDefault: Boolean = true,
  )(implicit traceContext: TraceContext): Future[DynamicDomainParameters] =
    findDynamicDomainParameters().map {
      case Right(value) => value.parameters
      case Left(_) =>
        if (warnOnUsingDefault) {
          logger.warn(s"Unexpectedly using default domain parameters at $
↪{timestamp}")
        }

        DynamicDomainParameters.initialValues(
          // we must use zero as default change delay parameter, as otherwise
↪static time tests will not work
          // however, once the domain has published the initial set of domain
↪parameters, the zero time will be
          // adjusted.
          topologyChangeDelay = DynamicDomainParameters.
↪topologyChangeDelayIfAbsent,
          protocolVersion = protocolVersion,
        )
    }

  def findDynamicDomainParameters()(implicit
      traceContext: TraceContext
  ): Future[Either[String, DynamicDomainParametersWithValidity]]

  /** List all the dynamic domain parameters (past and current) */
  def listDynamicDomainParametersChanges()(implicit
      traceContext: TraceContext
  ): Future[Seq[DynamicDomainParametersWithValidity]]
}

trait MembersTopologySnapshotClient {
  this: BaseTopologySnapshotClient =>

  def allMembers(): Future[Set[Member]]

  def isMemberKnown(member: Member): Future[Boolean]
```

(continues on next page)

(continued from previous page)

```
}

trait TopologySnapshot
    extends PartyTopologySnapshotClient
    with BaseTopologySnapshotClient
    with ParticipantTopologySnapshotClient
    with KeyTopologySnapshotClient
    with CertificateSnapshotClient
    with VettedPackagesSnapshotClient
    with MediatorDomainStateClient
    with SequencerDomainStateClient
    with DomainTrafficControlStateClient
    with DomainGovernanceSnapshotClient
    with MembersTopologySnapshotClient { this: BaseTopologySnapshotClient with
↪NamedLogging => }
```

Based on this API, the following Sync Crypto API can be built, which allows to decouple the crypto operations used in the synchronisation protocol from the crypto protocol and identity management implementation.

Sync Crypto Api

Within Canton, the entire identity, key and signing management is abstracted and hidden from the synchronisation protocol behind the *SyncCryptoApi*.

```
/** impure part of the crypto api with access to private key store and knowledge
↪about the current entity to key assoc */
trait SyncCryptoApi {

  def pureCrypto: CryptoPureApi

  def ipsSnapshot: TopologySnapshot

  /** Signs the given hash using the private signing key. */
  def sign(hash: Hash)(implicit
      traceContext: TraceContext
  ): EitherT[Future, SyncCryptoError, Signature]

  /** Decrypts a message using the private key of the public key given as the
↪fingerprint. */
  def decrypt[M](encryptedMessage: AsymmetricEncrypted[M])(
      deserialize: ByteString => Either[DeserializationError, M]
  )(implicit traceContext: TraceContext): EitherT[Future, SyncCryptoError, M]

  @Deprecated
  def decrypt[M](encryptedMessage: Encrypted[M])(
      deserialize: ByteString => Either[DeserializationError, M]
  )(implicit traceContext: TraceContext): EitherT[Future, SyncCryptoError, M]

  /** Verify signature of a given owner
    *
    * Convenience method to lookup a key of a given owner, domain and timestamp
↪and verify the result.
    */
```

(continues on next page)

(continued from previous page)

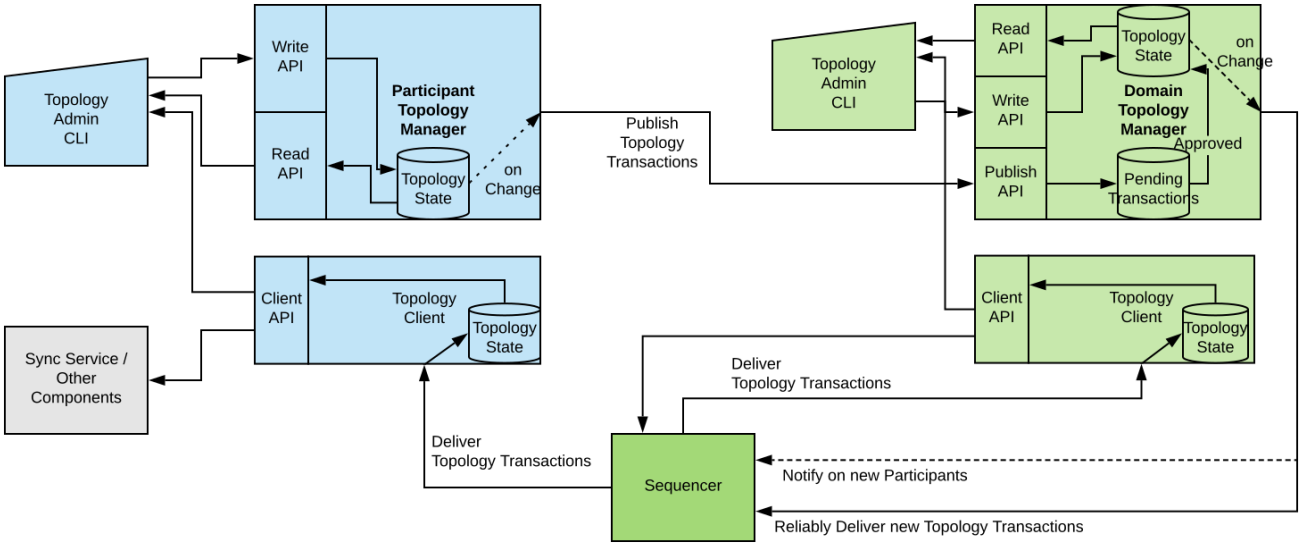
```
  def verifySignature(
      hash: Hash,
      signer: KeyOwner,
      signature: Signature,
  ): EitherT[Future, SignatureCheckError, Unit]

  def verifySignatures(
      hash: Hash,
      signer: KeyOwner,
      signatures: NonEmpty[Seq[Signature]],
  ): EitherT[Future, SignatureCheckError, Unit]

  /** Verifies a list of `signatures` to be produced by active members of a
↪`mediatorGroup`,
    * counting each member's signature only once.
    * Returns `Right` when the `mediatorGroup`'s threshold is met.
    * Can be successful even if some signatures fail the check, logs the errors
↪in that case.
    * When the threshold is not met returns `Left` with all the signature check
↪errors.
    */
  def verifySignatures(
      hash: Hash,
      mediatorGroupIndex: MediatorGroupIndex,
      signatures: NonEmpty[Seq[Signature]],
  )(implicit traceContext: TraceContext): EitherT[Future, SignatureCheckError,
↪Unit]

  /** Encrypts a message for the given key owner
    *
    * Utility method to lookup a key on an IPS snapshot and then encrypt the
↪given message with the
    * most suitable key for the respective key owner.
    */
  def encryptFor[M <: HasVersionedToByteString](
      message: M,
      owner: KeyOwner,
      version: ProtocolVersion,
  ): EitherT[Future, SyncCryptoError, AsymmetricEncrypted[M]]
}
```

This class contains the appropriate methods in order to *sign*, *verify signatures*, *encrypt* or *decrypt* on a per member basis. Which key and which cryptographic method is used is hidden entirely behind this API.

The API is obtained on a per domain and timestamp basis. The *SyncCryptoApiProvider* combines the information about the owner of the node, the connected domain, the cryptographic module in use and the topology state for a particular time and provides a factory method to obtain the *SyncCryptoApi* for a particular domain and time combination.

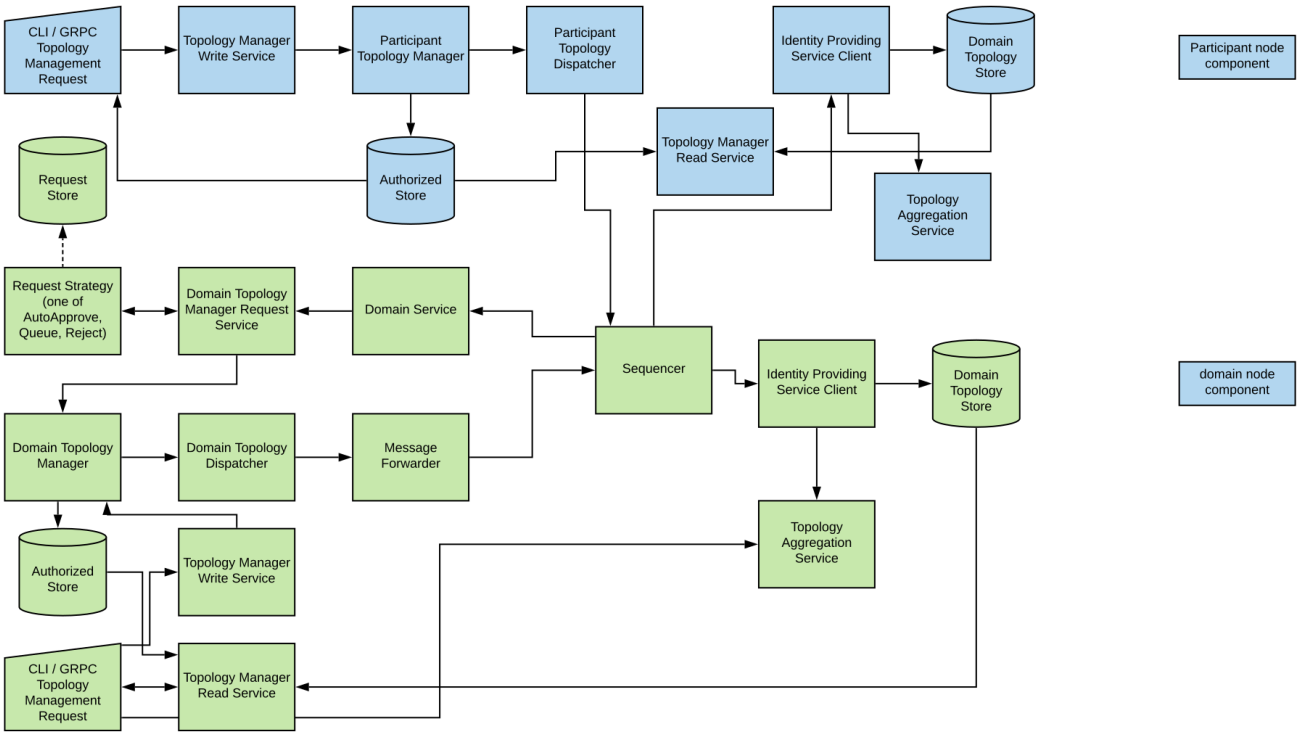
High-Level Picture

The following drawing provides a high-level overview of the identity management architecture and flows.



Transaction Flow

The following chart lays out all components of the Canton identity management system. Some of the components are shared between participant node and domain node, while some have slightly different functionality. The arrow indicates data flow.



In the following, we describe how a topology command invoked on the participant node propagates through the system. Ultimately, the component fully describing the topology state is the topology

providing service client (TPSC). Therefore, we can track the propagation from the command until it reaches the IPSC.

CLI/gRPC Topology Management Request - The topology management system is accessible through the *topology_manager_write_service*, the *topology_manager_read_service* and the *topology_aggregation_service*, which are gRPC based services. The Canton shell exposes all these services directly through appropriate commands.
Topology Manager Write Service - In order to change the topology state, an administrator needs to create a new topology transaction and authorize it by signing it with an eligible key. These authorization commands are externally accessible using the write service, exposing the gRPC API.
Participant Topology Manager - Every participant has a local topology manager. The participant can populate the store either by importing authorized transactions or creating new authorized transactions itself. The topology manager checks every locally added transaction for consistency and correctness.
Participant Topology Dispatcher - The dispatcher monitors the topology state managed by the local topology manager and tries to push the local authorized topology state to any connected domain. As an example, if a party is added locally, the dispatcher tries to propagate the corresponding topology transaction to any connected domain.
Sequencer Connect Service - Every sequencer exposes a public service, called sequencer connect service, for handshake and administrative purposes. Here, participants obtain the applicable domain rules, the protocol version and the domain id.
Domain Topology Manager Request Service - Any topology transaction upload from the domain service is processed through the request service. The request service is configured with a **request strategy**. The request strategy inspects the topology transaction and decides how to deal with a topology transaction. Right now, three strategies have been implemented: auto-approve for un-permissioned domains, queue for permissioned domains (where transactions are just stored for later decision in the *Request Store*) and reject for closed domains.
Domain Topology Manager - Similar to the participant node topology manager, except with the added functionality required for a domain, allowing it to set participant states. Changes to the domain topology manager either come from the local administrator through the topology manager write service or through accepted topology transactions from the request service. The sequencer listens to the domain topology manager and sets up new member queues if a new participant is added to the system.
Domain Topology Dispatcher - The domain topology dispatcher monitors the local authorized domain topology state. Upon a change, the dispatcher computes who needs to be informed of the given topology transaction (i.e. all active participant nodes). Or, if a new participant has been added, the dispatcher ensures that the first transactions a new participant will observe when connecting to the sequencer are the topology transactions. This prevents any race-condition or inconsistent topology state.
Message Forwarder - The topology state requires that the topology transactions are applied in the previously established order. The message forwarder therefore ensures the absolute guaranteed in order delivery of all topology transactions, in particular in the case of temporary delivery to sequencer failure. The message forwarder sends the topology transactions as instructed by the dispatcher via the sequencer to all participant nodes and domain entities.
Identity Providing Service Client - The implementation of the IPSC listens to the stream of sequenced messages and receives the identity updates. The client inspects the message, validates the signatures and appends the topology transaction to the topology state.
Topology Aggregation Service - Inspect via gRPC the aggregated topology state as exposed by the IPSC internally.

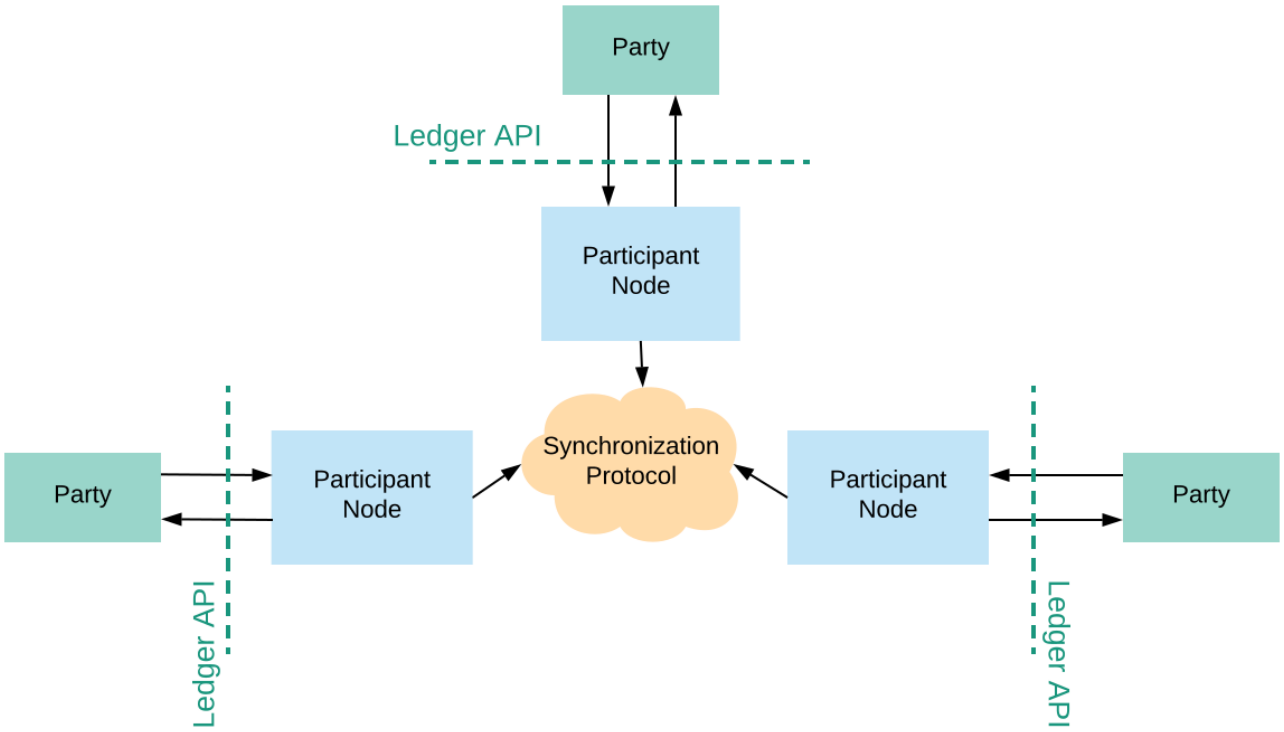
Not direct part of the transaction flow, but essential components for topology management are the

following components:

Authorized/Request/Domain Topology Store - There are several stores for topology transactions. The authorized store is the set of topology transactions that have been added to the local topology manager. The domain topology store is the store of topology transactions that have been timestamped by the sequencer. The authorized store of a domain and the domain topology store contain the same content, except that the authorized store can hold data which has not yet been timestamped by the sequencer. The content of the domain topology stores on the participant (one per connected domain) is exactly the same among all participants on a domain. These stores are used by the synchronisation protocol.
Topology Manager Read Service - The topology manager read service just serves inspection purposes in order to look deeply into the topology state. The read service plugs directly into a topology store and exposes the content via gRPC.

1.44.2.4 High-Level Requirements

As detailed in the *DA ledger model*, the Daml ledger interoperability protocol provides parties with a *virtual shared ledger*, which contains their interaction history and the current state of their shared Daml contracts. To access the ledger, the parties must deploy (or have someone deploy for them) the so-called *participant nodes*. The participant nodes then expose the *Ledger API*, which enables the parties to request changes and get notified about the changes to the ledger. To apply the changes, the participant nodes run a synchronization protocol. We can visualize the setup as follows.



In general, the setup might be more complicated than shown above, as a single participant node can provide services for more than one party and parties can be hosted on multiple participant nodes. Note, however, that this feature is currently limited. In particular, a party hosted on multiple participants should be on-boarded on all of them before participating to any transaction.

In this section, we list the high-level functional requirements on the Ledger API, as well as non-functional requirements on the synchronization protocol.

Functional requirements

Functional requirements specify the constraints on and between the system's observable outputs and inputs. A difficulty in specifying the requirements for the synchronization service is that the system and its inputs and outputs are distributed, and that the system can include **Byzantine** participant nodes, i.e., participants that are malicious, malfunctioning or compromised. The system does not have to give any guarantees to parties using such nodes, beyond the ability to recover from malfunction/compromise. However, the system must protect the **honestly represented parties** (i.e., parties all of whose participant nodes implement the synchronization service correctly) from malicious behavior. To account for this in our requirements, we exploit the fact that the conceptual purpose of the ledger synchronization service is to provide parties with a virtual shared ledger and we:

1. use such a shared ledger and the associated properties (described in the *DA ledger model*) to constrain the input-output relation;
2. express all requirements from the perspective of an honestly represented party;
3. use the same shared ledger for all parties and requirements, guaranteeing synchronization.

We express the high-level functional requirements as user stories, always from the perspective of an honestly represented party, i.e., Ledger API user, and thus omit the role. As the observable inputs and outputs, we take the Ledger API inputs and outputs. Additionally, we assume that crashes and recoveries of participant nodes are observable. The requirements ensure that the virtual shared ledger describes a world that is compatible with the honestly represented parties' perspectives, but it may deviate in any respect from what Byzantine nodes present to their parties. We call such parties **dishonestly represented parties**.

Some requirements have explicit exceptions and design limitations. Exceptions are fundamental, and cannot be improved on by further design iterations. Design limitations refer to the design of the Canton synchronization protocol and can be improved in future versions. We discuss the consequences of the most important exceptions and design limitations *later in the section*.

Note: The fulfillment of these requirements is conditional on the system's assumptions (in particular, any trusted participants must behave correctly).

Synchronization. I want the platform to provide a virtual ledger (according to the *DA ledger model*) that is shared with all other parties in the system (honestly represented or not), so that I stay synchronized with my counterparties.
Change requests possible. I want to be able to submit change requests to the shared ledger.
Change request identification. I want to be able to assign an identifier to all my change requests.
Change request deduplication. I want the system to deduplicate my change requests with the same identifiers, if they are submitted within a time window configurable per participant, so that my applications can resend change requests in case of a restart without adding the changes to the ledger twice.
Bounded decision time. I want to be able to learn within some bounded time from the submission (on the order of minutes) the decision about my change request, i.e., whether it was added to the ledger or not.
Design limitation: If the participant node used for the submission crashes, the bound can be exceeded. In this case the application should fail over to another participant replica.
Consensus. I want that all honestly represented counterparties come to the same conclusion of either accepting or rejecting a transaction according to the *DA ledger model*.

Transparency. I want to get notified in a timely fashion (on the order of seconds) about the changes to my *projection of the shared ledger*, according to the DA ledger model, so that I stay synchronized with my counterparties.
Design limitation: If the participant has crashed, is overloaded, or in case of network failures, the bound can be exceeded. In the case of a crash the application should fail over to another participant replica.
Design limitation: If the submitter node is malicious it can encrypt the transaction payload with the wrong key that my participant cannot decrypt it. I will be notified about a transaction but able to see its contents.
Integrity: ledger validity. I want the shared ledger to be *valid according to the DA ledger model*.
Exception: The *consistency* aspect of the validity requirement on the shared ledger can be violated for contracts with no honestly represented signatories, even if I am an observer on the contract.
Integrity: request authenticity. I want the shared ledger to contain a record of a change with me as one of the requesters if and only if:

1. I actually requested that exact change, i.e., I submitted the change via the command submission service, and
2. I am notified that my change request was added to the shared ledger, unless my participant node crashes forever,

so that, together with the ledger validity requirement, I can be sure that the ledger contains no records of:

1. obligations imposed on me,
2. rights taken away from me, and
3. my counterparties removing their existing obligations

without my explicit consent. In particular, I am the only requester of any such change. Note that this requirement implies that the change is done atomically, i.e. either it is added in its entirety, or not at all.
Remark: As functional requirements apply only to honestly represented parties, any dishonestly represented party can be a requester of a commit on the virtual shared ledger, even if it has never submitted a command via the command submission service. However, this is possible only if **no** requester of the commit is honestly represented.

Note: The two integrity requirements come with further limitations and trust assumptions, whenever the *trust-liveness trade-offs* below are used.

Non-repudiation. I want the service to provide me with irrefutable evidence of all ledger changes that I get notified about, so that I can prove to a third party (e.g., a court) that a contract of which I am a stakeholder was created or archived at a certain point in time.
This item is scheduled on the Daml roadmap.
Finality. I want the shared ledger to be append-only, so that, once I am notified about a change to the ledger, that change cannot be removed from the ledger.
Daml package uploads. I want to be able to upload a new Daml package to my participant node, so that I can start using new Daml contract templates or upgraded versions of existing ones. The authority to upload packages can be limited to particular parties (e.g., a participant administrator party), or done through a separate API.
Daml package notification. I want to be able to get notified about new packages distributed to me by other parties in the system, so that I can inspect the contents of the package, either automatically or manually.
Automated Daml package distribution. I want the system to be able to notify counterparties about my uploaded Daml packages before the first time that I submit a change request that includes a contract that both comes from this new package and has the counterparty as a

stakeholder on it.
Limitation The package distribution does not happen automatically on first use of a package, because manual (*Daml package vetting*) would lead to a rejection of the transaction.
Daml package vetting. I want to be able to explicitly approve (manually or automatically, e.g., based on a signature by a trusted party) every new package sent to me by another party, so that the participant node does not execute any code that has not been approved. The authority to vet packages can be limited to particular parties, or done through a separate API.
Exception: I cannot approve a package without approving all of its dependencies first.
No unnecessary rejections. I want the system to add all my well-authorized and Daml-conformant change requests to the ledger, unless:

1. they are duplicated, or
2. they use Daml templates my counterparties' participants have not vetted, or
3. they conflict with other changes that are included in the shared ledger prior to or at approximately the same time as my request, or
4. the processing capacity of my participant node or the participant nodes of my counterparties is exhausted by other change requests submitted by myself or others roughly simultaneously,

in which case I want the decision to include the appropriate reason for rejection.
Exception 1: This requirement may be violated whenever my participant node crashes, or if there is contention in the system (multiple conflicting requests are issued in a short period of time). The rejection reason reported in the decision in the exceptional case must differ from those reported because of other causes listed in this requirement.
Exception 2: If my change request contains an exercise on a contract identifier, and I have not witnessed (e.g., through *divulgence*) any actions on a contract with this identifier in my *projection of the shared ledger* (according to the DA ledger model), then my change request may fail.
Design limitation 1: My change requests can also be rejected if a participant of some counterparty (hosting a signatory or an observer) in my change request is crashed, unless some trusted participant (e.g., one run by a market operator) is a stakeholder participant on all contracts in my change request.
Design limitation 2: My change requests can also be rejected if any of my counterparties in the change request is Byzantine, unless some trusted participant (e.g., one run by a market operator) is a stakeholder participant on all contracts in my change request.
Design limitation 3: If the underlying sequencer queue is full for a participant, then we can get an unnecessary rejection. We assume however that the queue size is so large that it can be considered to be infinite, so this unnecessary rejection doesn't happen in practice, and the situation would be resolved operationally before the queue fills up.
Design limitation 4: If the mediator of the domain has crashed and lost the in-flight transaction, which will then timeout.
Seek support for notifications. I want to be able to receive notifications (about ledger changes and about the decisions on my change requests) only from a particular known offset, so that I can synchronize my application state with the set of active contracts on the shared ledger after crashes and other events, without having to read all historical changes.
Exception: A participant can define a bound on how far in the past the seek can be requested.
Active contract snapshots. I want the system to provide me a way to obtain a recent (on the order of seconds) snapshot of the set of active contracts on the shared ledger, so that I can initialize my application state and synchronize with the set of active contracts on the ledger efficiently.
Change request processing limited to participant nodes. I want only the following (and no other) functionality related to change request processing:

1. submitting change requests
2. receiving information about change request processing and results

3. (possibly) vetting Daml packages

to be exposed on the Ledger API, so that the unavailability of my or my counterparties' applications cannot influence whether a change I previously requested through the API is included in the shared ledger, except if the request is using packages not previously vetted. Note that this inclusion may still be influenced by the availability of my counterparties' participant nodes (as specified in the limitations on the *requirement on no unnecessary rejections*)

Resource limits

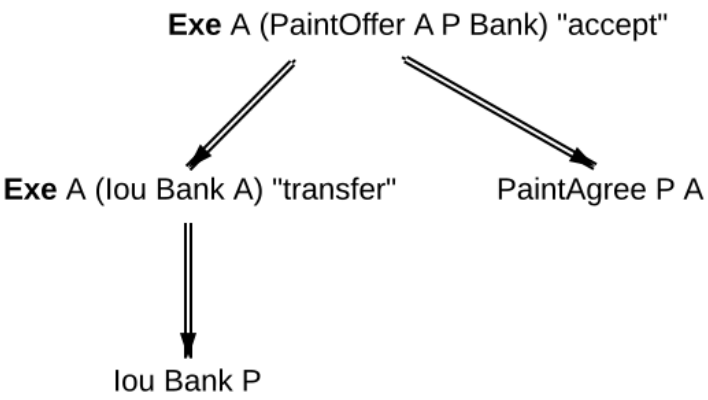
This section specifies upper bounds on the sizes of data structures. The system must be able to process data structures within the given size limits.

If a data structure exceeds a limit, the system must reject transactions containing the data structure. Note that it would be impossible to check violations of resource limits at compile time; therefore the Daml compiler will not emit an error or warning if a resource limit is violated.

Maximum transaction depth: 100

Definition: The maximum number of levels (except for the top-level) in a transaction tree.

Example: The following transaction has a depth of 2:



Purpose: This limit is to mitigate the higher cost of implementing stack-safe algorithms on transaction trees. The limit may be relaxed in future versions.

Maximum depth of a Daml value: 100

Definition: The maximum numbers of nestings in a Daml value.

Example:

The value 17 has a depth of 0.
The value {myField: 17} has a depth of 1.
The value [{myField: 17}] has a depth of 2.
The value ['observer1', 'observer2', , 'observer100'] has a depth of 1.

Purpose:

1. Applications interfacing the DA ledger likely have to process Daml values and likely are developed outside of DA. By limiting the depth of Daml values, application devel-

opers have to be less concerned about stack usage of their applications. So the limit effectively facilitates the development of applications.
2. This limit allows for a readable wire format for Daml-LF values, as it is not necessary to flatten values before transmission.

Non-functional requirements

These requirements specify the characteristics of the internal system operation. In addition to the participant nodes, the implementation of the synchronization protocol may involve a set of additional operational entities. For example, this set can include a sequencer. We call a single deployment of such a set of operational entities a **domain**, and refer to the entities as **domain entities**.

As before, the requirements are expressed as user stories, with the user always being the Ledger API user. Additionally, we list specific requirements for financial market infrastructure providers. Some requirements have explicit exceptions; we discuss the consequences of these exceptions *later in the section*.

Privacy. I want the visibility of the ledger contents to be restricted according to the *privacy model of DA ledgers*, so that the information about any (sub)action on the ledger is provided only to participant nodes of parties privy to this action. In particular, other participant nodes must not receive any information about the action, not even in an encrypted form.
Exception: domain entities operated by trusted third parties (such as market operators) may receive encrypted versions of any of the ledger data (but not plain text).
Design limitation 1: Participant nodes of parties privy to an action (according to the ledger privacy model) may learn the following:
- How deeply lies the action within a ledger commit.
- How many sibling actions each parent action has.
- The transaction identifiers (but not the transactions' contents) that have created the contracts used by the action.

Design limitation 2: Domain entities operated by trusted third parties may learn the hierarchical structure and stakeholders of all actions of the ledger (but none of the contents of the contracts, such as templates used or their arguments).
Transaction stream auditability. I want the system to be able to convince a third party (e.g., an auditor) that they have been presented with my complete transaction stream within a configurable time period (on the order of years), so that they can be sure that the stream represents a complete record of my ledger projection, with no omissions or additions.
Exception: The evidence can be linear in the size of my transaction stream.
Design limitation: The evidence need not be privacy-preserving with respect to other parties with whom I share participant nodes, and the process can be manual.
This item is scheduled on the Daml roadmap.

Service Auditability. I want the synchronization protocol implementation to store all requests and responses of all participant nodes within a configurable time period (on the order of years), so that an independent third party can manually audit the correct behavior of any individual participant and ensure that all requests and responses it sent comply with the protocol.
Remarks The system operator has to make a trade-off between preserving data for auditability and deleting data for system efficiency (see *Pruning*).
This item is scheduled on the Daml roadmap.

GDPR Compliance. I want the system to be compliant with the General Data Protection Regulation (GDPR).

Configurable trust-liveness trade-off. I want each domain to allow me to choose from a predefined (by the domain) set of trade-offs between trust and liveness for my change requests, so that my change requests get included in the ledger even if some of the participant nodes of my counterparties are offline or Byzantine, at the expense of making additional trust assumptions: on (1) the domain entities (for privacy and integrity), and/or (2) participant nodes run by counterparties in my change request that are marked as VIP by the domain (for integrity), and/or (3) participant nodes run by other counterparties in my change request (also for integrity).
Exception: If the honest and online participants do not have sufficient information about the activeness of the contracts used by my change request, the request can still be rejected.
Design limitation: The only trade-off allowed by the current design is through confirmation policies. Currently, the only fully supported policies are the full, signatory, and VIP confirmation policies. The implementation does not support the serialization of other policies. Furthermore, integrity need not hold under other policies. This corresponds to allowing only the trade-off (2) above (making additional trust assumptions on VIP participants). In this case, the VIP participants must be trusted.
Note: If a participant is trusted, then the trust assumption extends to all parties hosted by the participant. Conversely, the system does not support to trust a participant for the actions performed on behalf of one party and distrust the same participant for the actions performed on behalf of a different party.
Workflow isolation. I want the system to be built such that workflows (groups of change requests serving a particular business purpose) that are independent, i.e. do not conflict with other, do not affect each other's performance.
This item is scheduled on the roadmap.

Pruning. I want the system to provide data pruning capabilities, so that the required hot storage capacity for each participant node depends only on:

1. the size of currently active contracts whose processing the node is involved in,
2. the node's past traffic volume within a (per-participant) configurable time window

and does not otherwise grow unboundedly as the system continues operating. Cold storage requirements are allowed to keep growing continuously with system operation, for auditability purposes.
Multi-domain participant nodes. I want to be able to use multiple domains simultaneously from the same participant node.
This item is only delivered in an experimental state and scheduled on the roadmap for GA.
Internal participant node domain. I want to be able to use an internal domain for workflows involving only local parties exclusively hosted by the participant node.
This item is scheduled on the roadmap.
Connecting to domains. I want to be able to connect my participant node to a new domain at any point in time, as long as I am accepted by the domain operators.
Exception If the participant has been connected to a domain with unique contract key mode turned on, then connecting to another domain is forbidden.
Workflow transfer. I want to be able to transfer the processing of any Daml contract that I am a stakeholder of or have delegation rights on, from one domain to another domain that has been vetted as appropriate by all contract stakeholders through some procedure defined by the synchronization service, so that I can use domains with better performance, do load balancing and disaster recovery.
Workflow composability. I want to be able to atomically execute steps (Daml actions) in different workflows across different domains, as long as there exists a single domain to which all participants in all workflows are connected.
This item is scheduled on the roadmap.

Standards compliant cryptography. I want the system to be built using configurable cryptographic primitives approved by standardization bodies such as NIST, so that I can rely on existing audits and hardware security module support for all the primitives.

Secure storage of cryptographic private keys. I want the system to store cryptographic private keys in a secure way to prevent unauthorized access to the keys.
Upgradability. I want to be able to upgrade system components, both individually and jointly, so that I can deploy fixes and improvements to the components and the protocol without stopping the system's operation.
Design Limitation 1 When a domain needs to be upgraded to a new protocol version a new domain is deployed and the participants migrate the active contracts' synchronization to the new domain.
Design Limitation 2 When a replicated node needs to be upgraded, all replicas of the node needs to be upgraded at the same time.

Semantic versioning. I want all interfaces, protocols and persistent data schemas to be versioned, so that version mismatches are prevented. The versioning scheme must be semantic, so that breaking changes always bump the major versions.
Remark Every change in the Canton protocol leads to a new major version of the protocol, but a Canton node can support multiple protocols without requiring a major version change.
Domain approved protocol versions. I want domains to specify the allowed set of protocol versions on the domain, so that old versions of the protocol can be decommissioned, and that new versions can be introduced and rolled back if operational problems are discovered.
Design limitation: Initially, the domain can specify only a single protocol version as allowed, which can change over time.
Multiple protocol compatibility. I want new versions of system components to still support at least one previous major version of the synchronization protocol, so that entities capable of using newer versions of the protocol can still use domains that specify only old versions as allowed.

Testability of participant node upgrades on historical data. I want to be able to test new versions of participant nodes against historical data from a time window and compare the results to those obtained from old versions, so that I can increase my certainty that the new version does not introduce unintended differences in behavior.
This item is scheduled on the roadmap.

Seamless participant failover. I want the applications using the ledger API to seamlessly fail over to my other participant nodes, once one of my nodes crashes.
Design limitation An external load balancer is required in front of the participant node replicas to direct requests to the appropriate replica.

Seamless failover for domain entities. I want the implementation of all domain entities to include seamless failover capabilities, so that the system can continue operating uninterruptedly on the failure of an instance of a domain entity.

Backups. I want to be able to periodically backup the system state (ledger databases) so that it can be subsequently restored if required for disaster recovery purposes.

Site-wide disaster recovery. I want the system to be built with the ability to recover from a failure of an entire data center by moving the operations to a different data center, without loss of data.

Participant corruption recovery. I want to have a procedure in place that can be followed to recover from a malfunctioning or a corrupted participant node, so that when the procedure is finished I obtain the same guarantees (in particular, integrity and transparency) as the honest

participants on the part of the shared ledger created after the end of the recovery procedure.

Domain entity corruption recovery. I want to have a procedure in place that can be followed to recover a malfunctioning or corrupted domain entity, so that the system guarantees can be restored after the procedure is complete.
This item is scheduled on the roadmap.

Fundamental dispute resolution. I want to have a procedure in place that allows me to limit and resolve the damage to the ledger state in the case of a fundamental dispute on the outcome of a transaction that was added to the virtual shared ledger, so that I can reconcile the set of active contracts with my counterparties in case of any disagreement over this set. Example causes of disagreement include disagreement with the state found after recovering a compromised participant, or disagreement due to a change in the regulatory environment making some existing contracts illegal.
This item is scheduled on the roadmap.

Distributed recovery of participant data. I want to be able to reconstruct which of my contracts are currently active from the information that the participants of my counterparties store, so that I can recover my data in case of a catastrophic event. This assumes that the other participants are cooperating and have not suffered catastrophic failures themselves.
This item is scheduled on the roadmap.

Adding parties to participants. I want to be able to start using the system at any point in time, by choosing to use a new or an already existing participant node.

Identity information updates. I want the synchronization protocol to track updates of the topology state, so that the parties can switch participants, and participants can roll and/or revoke keys, while ensuring continuous system operation.
Party migration. I want to be able to switch from using one participant node to using another participant node, without losing the data about the set of active contracts on the shared ledger that I am a stakeholder of. The new participant node does not need to provide me with the ledger changes prior to migration.
This item is scheduled on the roadmap.

Parties using multiple participants. I want to be able to use the system through multiple participant nodes, so that I can do load balancing, and continue using the system even if one of my participant nodes crashes.
Design limitation The usage of multiple participants by a single party is not seamless as with *participant high availability*, because ledger offsets are different between participant nodes unless it is a replicated participant and command deduplication state is not shared among multiple participant nodes.
Read-only participants. I want to be able to configure some participants as read-only, so that I can provide a live stream of the changes to my ledger view to an auditor, without giving them the ability to submit change requests.
Reuse of off-the-shelf solutions. I want the system to rely on industry-standard abstractions for:

1. messaging
2. persistent storage (e.g., SQL)
3. identity providers (e.g., Oauth)
4. metrics (e.g., MetricsRegistry)
5. logging (e.g., Logback)
6. monitoring (e.g., exposing `/health` endpoints)

so that I can use off-the-shelf solutions for these purposes.

Metrics on APIs. I want the system to provide metrics on the state of all API endpoints in the

system, and make them available on both link endpoints.

Metrics on processing. I want the system to provide metrics for every major processing phase within the system.

Component health monitoring. I want the system to provide monitoring information for every system component, so that I am alerted when a component fails.
This item is scheduled on the roadmap.

Remote debugability. I want the system to capture sufficient information such that I can debug remotely and post-mortem any issue in environments that are not within my control (OP).

Horizontal scalability. I want the system to be able to horizontally scale all parallelizable parts of the system, by adding processing units for these parts.
This item is scheduled on the roadmap.

Large transaction support. I want the system to support large transactions such that I can guarantee atomicity of large scale workflows.
This item is scheduled on the roadmap.

Resilience to erroneous behavior. I want that the system to be resilient against erroneous behavior of users and participants such that I can entrust the system to handle my business.
This item is scheduled on the roadmap.

Resilience to faulty domain behavior. I want that the system to be able to detect and recover from faulty behaviour of domain components, such that occasional issues don't break the system permanently.
This item is scheduled on the roadmap.

Known limitations

In this section, we explain current limitations of Canton that we intend to overcome in future versions.

Limitations that apply always

Missing Key features

Cross-domain transactions currently require the submitter of the transaction to transfer all used contracts to a common domain. Cross-domain transactions without first transferring to a single domain are not supported yet. Only the stakeholders of a contract may transfer the contract to a different domain. Therefore, if a transaction spans several domains and makes use of delegation to non-stakeholders, the submitter currently needs to coordinate with other participants to run the transaction, because the submitter by itself cannot transfer all used contracts to a single domain.

Reliability

H2 support: The H2 database backend is not supported for production scenarios, therefore data continuity is also not guaranteed.

Manageability

Party migration is still an experimental feature. A party can already be migrated to a fresh participant that has not yet been connected to any domains. Party migration is currently a manual process that needs to be executed with some care.

Security

Denial of service attacks: We have not yet implemented all countermeasures to denial of service attacks. However the domain already protects against faulty participants sending too many requests and message size limits protect against malicious participants trying to send large amounts of data via a domain. Further rate limit on the ledger API protects against malicious/faulty applications.
Public identity information: The topology state of a domain (i.e., participants known to the domain and parties hosted by them) is known to all participants connected to the domain.

Limitations that apply only sometimes

Manageability

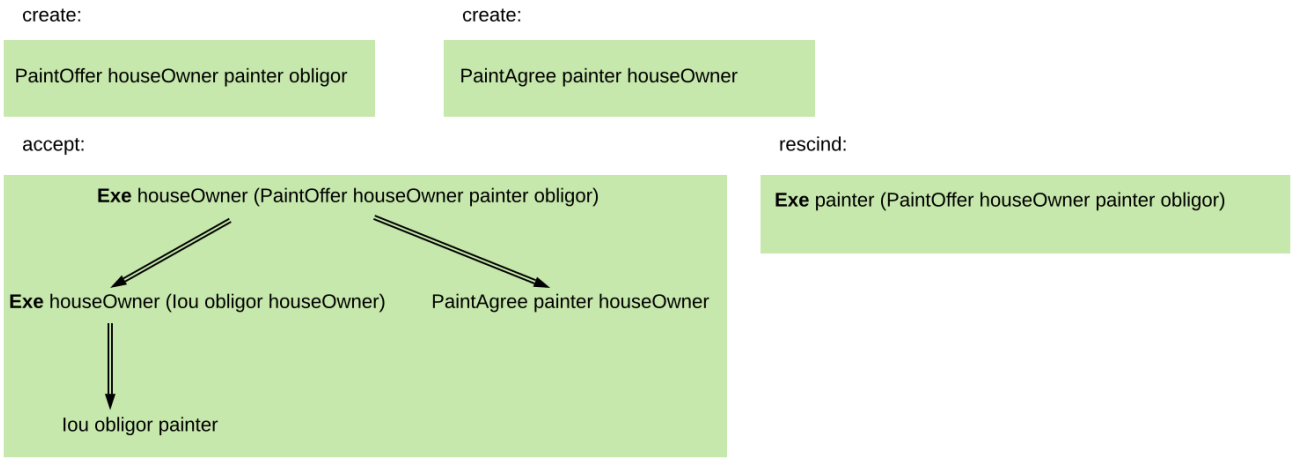
Multi-participant parties: Hosting a party on several participants is an experimental feature. If such a party is involved in a contract transfer, the transfer may result in a ledger fork, because the ledger API is not able to represent the situation that a contract is transferred out of scope of a participant. If one of the participants hosting a party is temporarily disabled, the participant may end up in an outdated state. The ledger API does not support managing parties hosted on several participants.
Disabling parties: If a party is disabled on a participant, it will remain visible on the ledger API of the participant, although it cannot be used anymore.
Pruning: The public API does not yet allow for pruning of contract transfers and rotated cryptographic keys. An offline participant can prevent the pruning of contracts by its counter-participants.
DAR and package management through the ledger API: A participant provides two APIs for managing DARs and Daml packages: the ledger API and the admin API. When a DAR is uploaded through the ledger API, only the contained packages can be retrieved through the admin API; the DAR itself cannot. When a package is uploaded through the ledger API, Canton needs to perform some asynchronous processing until the package is ready to use. The ledger API does not allow for querying whether a package is ready to use. Therefore, the admin API should be preferred for managing DARs and packages.

Requirement Exceptions: Notes

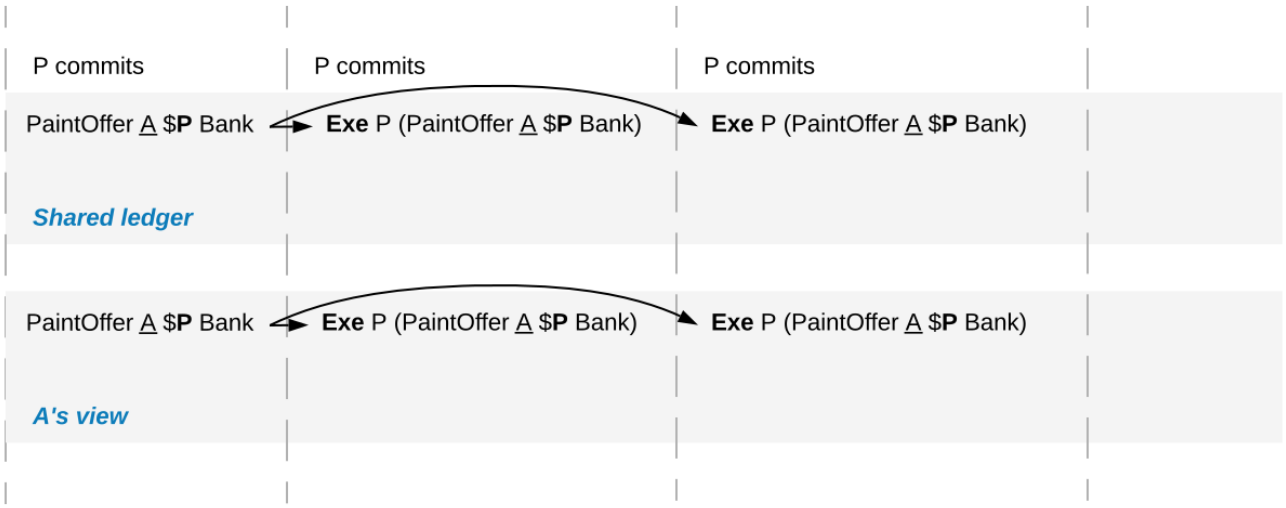
In this section, we explain the consequences of the exceptions to the requirements. In contrast to the *known limitations*, a requirements exception is a fundamental limitation of Canton that will most likely not be overcome in the foreseeable future.

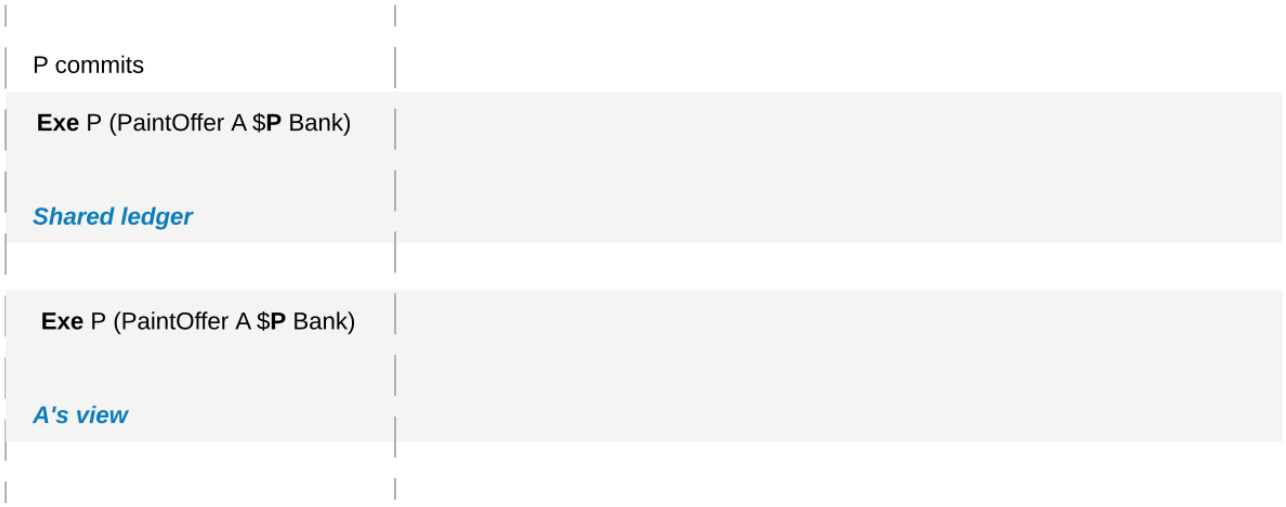
Ledger consistency

The validity requirement on the ledger made an exception for the *consistency* of contracts without honestly represented signatories. We explain the exception using the paint offer example from the *ledger model*. Recall that the example assumed contracts of the form *PaintOffer houseOwner painter obligor* with the *painter* as the signatory, and the *houseOwner* as an observer (while the obligor is not a stakeholder). Additionally, assume that we extend the model with an action that allows the painter to rescind the offer. The resulting model is then:



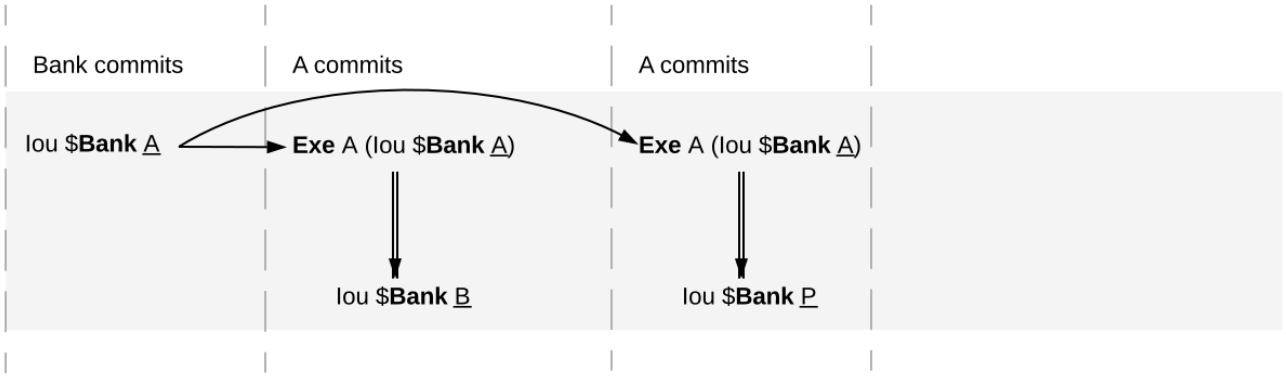
Assume that Alice (*A*) is the house owner, *P* the painter, and that the painter is dishonestly represented, in that he employs a malicious participant, while Alice is honestly represented. Then, the following shared ledgers are allowed, together with their projections for A, which in this case are just the list of transactions in the shared ledger.





That is, the dishonestly represented painter can rescind the offer twice in the shared ledger, even though the offer is not active any more by the time it is rescinded (and thus consumed) for the second time, violating the consistency criterion. Similarly, the dishonestly represented painter can rescind an offer that was never created in the first place.

However, this exception is not a problem for the stated benefits of the integrity requirement, as the resulting ledgers still ensure that honestly represented parties cannot have obligations imposed on them or rights taken away from them, and that their counterparties cannot escape their existing obligations. For instance, the example of a malicious Alice double spending her IOU:



is still disallowed even under the exception, as long as the bank is honestly represented. If the bank was dishonestly represented, then the double spend would be possible. But the bank would not gain anything by this dishonest behavior – it would just incur more obligations.

No unnecessary rejections

This requirement made exceptions for (1) contention, and included a design limitation for (2) crashes/Byzantine behavior of participant nodes. Contention is a fundamental limitation, given the requirement for a bounded decision time. Consider a sequence $cr_1, \ldots cr_n$ of change requests, each of which conflicts with the previous one, but otherwise have no conflicts, except for maybe cr_1. Then all the odd-numbered requests should get added to the ledger exactly when cr_1 is added, and the even-numbered ones exactly when cr_1 is rejected. Since detecting conflicts and other forms of processing (e.g. communication, Daml interpretation) incur processing delays, deciding precisely whether cr_n gets added to the ledger takes time proportional to n. By lengthening the sequence of requests, we eventually exceed any fixed bound within which we must decide on cr_n.

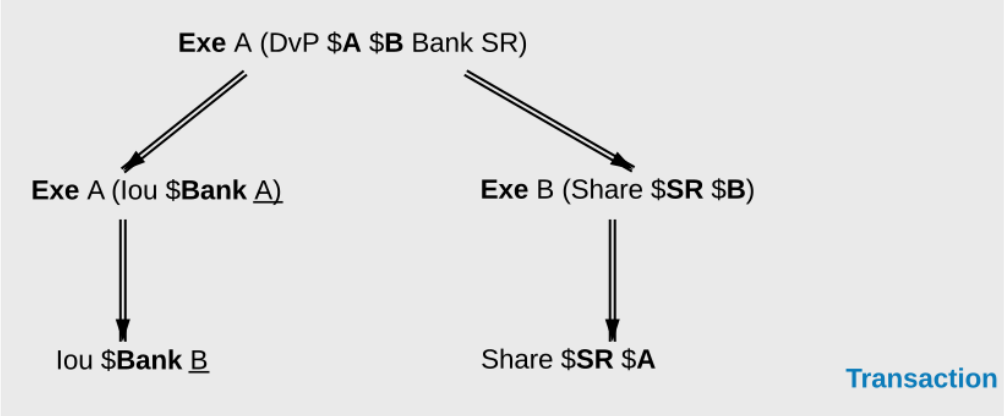
Crashes and Byzantine behavior can inhibit liveness. To cope, the so-called VIP confirmation policy allows any trusted participant to add change requests to the ledger without the involvement of other parties. This policy can be used in settings where there is a central trusted party. Today's financial markets are an example of such a setting.

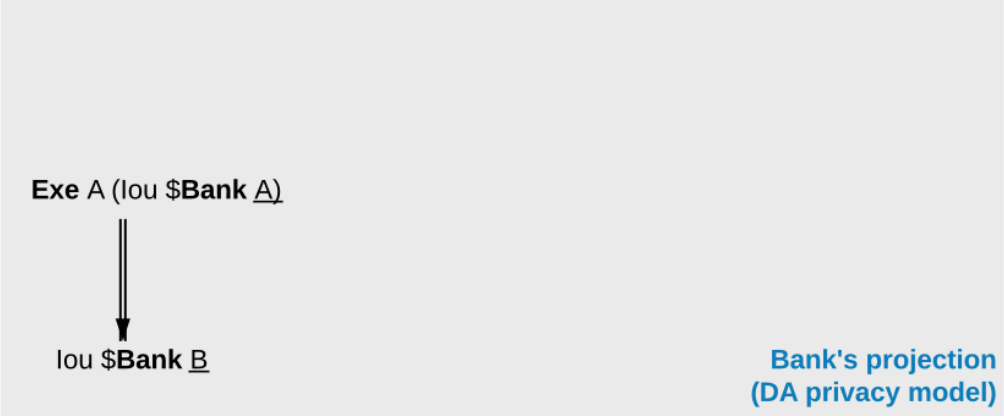
The no-rejection guarantees can be further improved by constructing Daml models that ensure that the submitter is a stakeholder on all contracts in a transaction. That way, rejects due to Byzantine behavior of other participants can be detected by the submitter. Furthermore, if necessary, the synchronization service itself could be changed to improve its properties in a future version, by including so-called bounded timeout extensions and attestators.

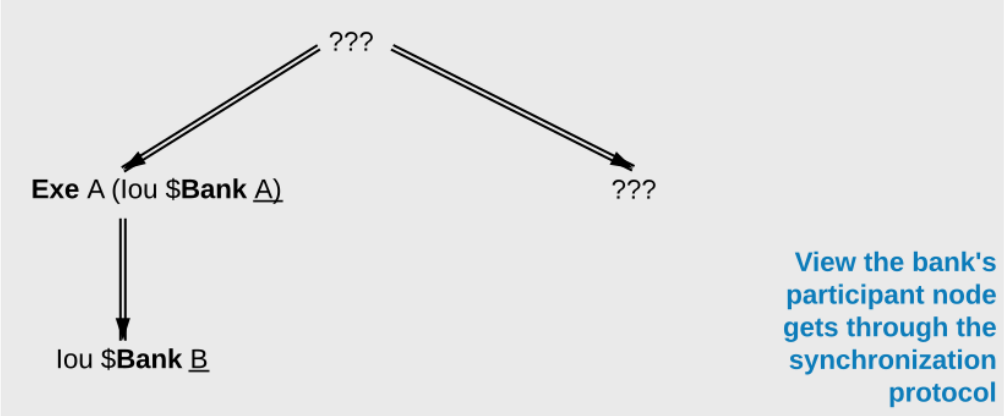
Privacy

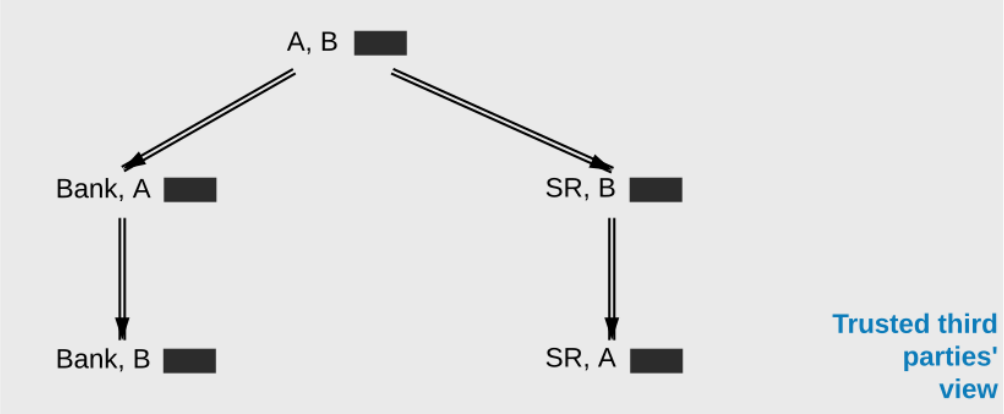
Consider a transaction where Alice buys some shares from Bob (a delivery-versus-payment transaction). The shares are registered at the share registry SR, and Alice is paying with an IOU issued to her by a bank. We depict the transaction in the first image below. Next, we show the bank's projection of this transaction, according to the DA ledger model. Below, we demonstrate what the bank's view obtained through the ledger synchronization protocol may look like. The bank sees that the transfer happens as a direct consequence of another action that has an additional consequence. However, the bank learns nothing else about the parent action or this other consequence. It does not learn that the parent action was on a DvP contract, that the other consequence is a transfer of shares, and that this consequence has further consequences. It learns neither the number nor the identities of the parties involved in any part of the transaction other than the IOU transfer. This illustrates the first design limitation for the privacy requirement.

At the bottom, we see that the domain entities run by a trusted third party can learn the complete structure of the transaction and the stakeholders of all actions in the transaction (second design limitations). Lastly, they also see some data about the contracts on which the actions are performed, but this data is visible *only in an encrypted form.* The decryption keys are never shared with the domain entities.


Exe A (DvP $A $B Bank SR)
Exe A (Iou $Bank A)
Exe B (Share $SR $B)
Iou $Bank B
Share $SR $A
Transaction


Exe A (Iou $Bank A)
Iou $Bank B
Bank's projection
(DA privacy model)


???
Exe A (Iou $Bank A)
???
Iou $Bank B
View the bank's
participant node
gets through the
synchronization
protocol


A, B
Bank, A
SR, B
Bank, B
SR, A
Trusted third
parties'
view

1.44.2.5 Research Publications

Daml, Canton, and their underlying theory are described in the following research publications:

Daml: A Smart Contract Language for Securely Automating Real-World Multi-Party Business Workflows describes the theory underlying Daml's language primitives for smart contracts and how Daml is compiled.
Alexander Bernauer, Sofia Faro, Rémy Hämmerle, Martin Huschenbett, Moritz Kiefer, Andreas Lochbihler, Jussi Mäki, Francesco Mazzoli, Simon Meier, Neil Mitchell, Ratko G. Veprek. *Daml: A Smart Contract Language for Securely Automating Real-World Multi-Party Business Workflows.* In: arXiv:2303.03749, 2023.
Abstract: Distributed ledger technologies, also known as blockchains for enterprises, promise to significantly reduce the high cost of automating multi-party business workflows. We argue that a programming language for writing such on-ledger logic should satisfy three desiderata:

1. Provide concepts to capture the legal rules that govern real-world business workflows.
2. Include simple means for specifying policies for access and authorization.
3. Support the composition of simple workflows into complex ones, even when the simple workflows have already been deployed.

We present the open-source smart contract language Daml based on Haskell with strict evaluation. Daml achieves these desiderata by offering novel primitives for representing, accessing, and modifying data on the ledger, which are mimicking the primitives of today's legal systems. Robust access and authorization policies are specified as part of these primitives, and Daml's built-in authorization rules enable delegation, which is key for workflow composability. These properties make Daml well-suited for orchestrating business workflows across multiple, otherwise heterogeneous parties.
Daml contracts run (1) on centralized ledgers backed by a database, (2) on distributed deployments with Byzantine fault tolerant consensus, and (3) on top of conventional blockchains, as a second layer via an atomic commit protocol.
A Structured Semantic Domain for Smart Contracts describes how Canton relates to Daml and the ledger model.
Extended abstract presented at Computer Security Foundations 2019.
Authenticated Data Structures As Functors in Isabelle/HOL formalizes Canton's Merkle tree data structures in the theorem prover Isabelle/HOL.

- Andreas Lochbihler and Ognjen Maric. *Authenticated Data Structures As Functors in Isabelle/HOL.* In: Bruno Bernardo and Diego Marmsoler (eds.) Formal Methods for Blockchain 2020. OASIcs vol. 84, 6:1-6:15, 2020.
- DOI
- Preprint PDF
- Pre-reecorded talk
- Live presentation (1:48 to 12:50)

A longer version was presented at the Isabelle Workshop 2020 (recording). The Isabelle theories are available in the Archive of Formal Proofs.
Abstract: Merkle trees are ubiquitous in blockchains and other distributed ledger technologies (DLTs). They guarantee that the involved systems are referring to the same binary tree, even if each of them knows only the cryptographic hash of the root. Inclusion proofs allow knowledgeable systems to share subtrees with other systems and the latter can verify the subtrees' authenticity. Often, blockchains and DLTs use data structures more complicated than binary trees; authenticated data structures generalize Merkle trees to such structures.
We show how to formally define and reason about authenticated data structures, their inclusion proofs, and operations thereon as datatypes in Isabelle/HOL. The construction lives in the symbolic model, i.e., we assume that no hash collisions occur. Our approach is modular and allows us to construct complicated trees from reusable building blocks, which we call Merkle

functors. Merkle functors include sums, products, and function spaces and are closed under composition and least fixpoints. As a practical application, we model the hierarchical transactions of Canton, a practical interoperability protocol for distributed ledgers, as authenticated data structures. This is a first step towards formalizing the Canton protocol and verifying its integrity and security guarantees.

A semantic domain for privacy-aware smart contracts and interoperable sharded ledgers Lightning talk presented at Certified Proofs and Programs 2021.

Abstract:

Daml is a Haskell-based smart contract programming language used to coordinate business workflows across trust boundaries. Daml's semantics are defined over an abstract ledger, which provides a clear semantics for Daml's authorization rules, double-spending protection, and privacy guarantees. In its simplest form, a ledger is represented as a list of commits, i.e., hierarchical transactions and their authorizers. This representation allows for easy reasoning about Daml smart contracts because the total order hides the intricacies of a distributed, Byzantine-fault tolerant system. It is also adequate for Daml running on a single blockchain, as it defines a total order on all transactions.

Yet, for distributed ledgers to fully eliminate data silos, smart contracts must not be tied to a single blockchain, which would then just become another silo. Daml therefore runs on different blockchains such as Hyperledger Fabric, Ethereum, and FISCO-BCOS as well as off-the-shelf databases. The underlying protocol Canton supports atomic transactions across all these Daml ledgers. This makes Daml ledgers sharded for higher throughput as well as interoperable to avoid data silos.

Semantically, Canton creates a virtual shared ledger by merging the individual ledgers' lists of commits. The virtual shared ledger is not totally ordered, to account for the fact that there is no global notion of time across ledgers. Still, transactions can use only contracts that have been created within earlier transactions. This ensures that causality is respected even though individual system users cannot see all dependencies due to the privacy rules. Canton tracks privacy-aware causality using vector clocks.

To ensure that Daml and Canton achieve their claimed properties, we have started to formalize the Daml ledger model and prove its properties in Isabelle/HOL. The two main verification goals are as follows:

1. Canton's vector clock tracking correctly implements causality.
2. The synchronization due to vector clocks cannot cause deadlocks.

The challenge here is that these guarantees should hold for honest nodes in the system even if other systems fail or behave Byzantine.

In the lightning talk, we give an idea of the ledger model, privacy-aware causality, and the current state of the verification.

1.44.2.6 Security Architecture

Secure Cryptographic Private Key Storage

In this section we describe Canton's two different approaches to securing the storage of cryptographic private keys. When enabled, we leverage a Key Management Service (KMS) to either: (a) *host an encryption key that is used to transparently encrypt the private keys (i.e. envelope encryption) before storing them in Canton's database*; or (b) *directly use a KMS to perform cryptographic operations without access to the private keys.* While using envelope encryption we make sure that an attacker who has access to the database (e.g., a malicious database operator) cannot get access to the private keys from a Canton node, which would compromise the transaction privacy and integrity guarantees of Canton. If we instead decide to externalize private key storage and usage, we go one step further and protect

against an attacker with privileged access to the node's system that can inspect the memory.

Background

Canton uses cryptography in several ways to provide a secure, minimal trust platform. In Canton we distinguish between three types of keys: short-term, long-term, and permanent keys.

Short-term key: These are used to encrypt transaction payloads. The secrets for these keys are already transmitted and stored in an encrypted form, and used only once.
Long-term key: These are keys that are used for topology transaction signing, transaction protocol signing, and encryption of short-term key secrets.
Permanent key: A namespace root signing key is a permanent key. It cannot be rotated without losing the namespace, as the namespace is identified by the fingerprint of the signing key. This is an architectural feature.

Long-term and permanent keys are by default stored in clear. Canton can, *if enabled*, offer confidentiality at rest for these private keys. Short-term keys do not require additional protection because they are derived from a secret that is already transmitted and stored in an encrypted form using a long-term public encryption key.

Long-term keys should be governed by an operational security policy with a requirement to rotate these keys periodically or if one of them is compromised.

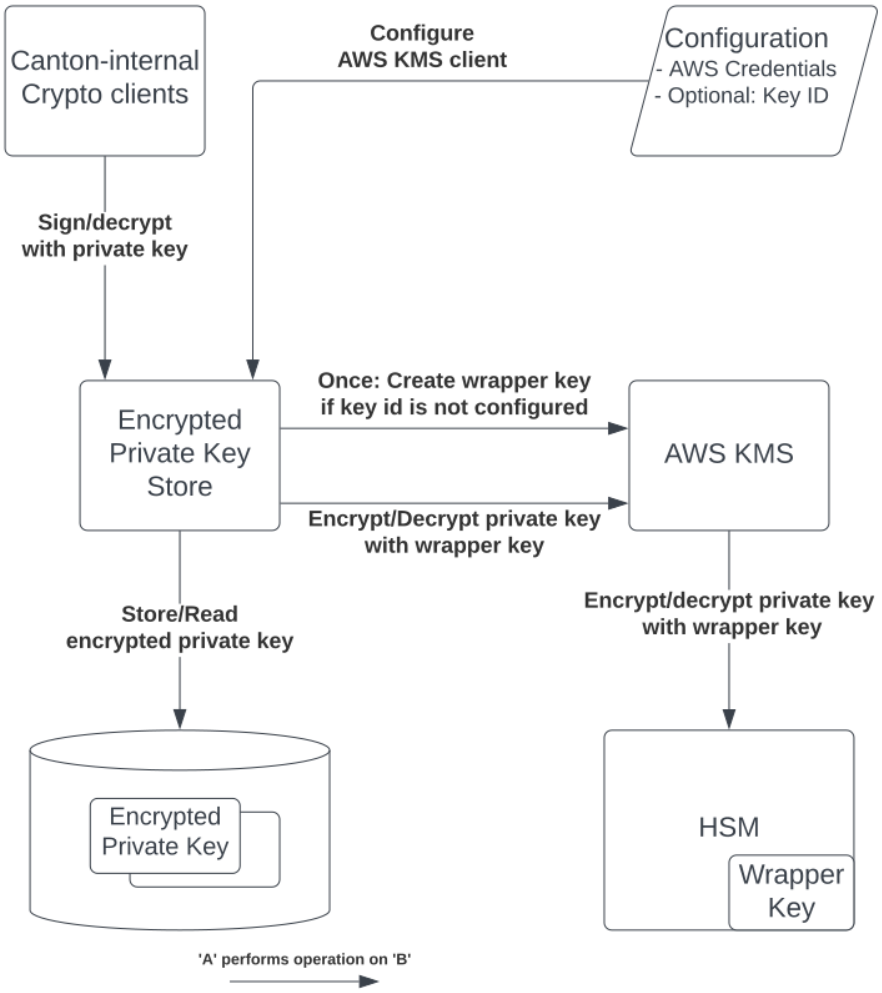
Requirements

The long-term keys must not be available on disk or in storage in a way that would allow someone with access to the storage to view/access the key.
The keys must not be part of Canton's container images.
A key administrator can rotate both the KMS key and the long-term keys in Canton.
Historical contract data can be decrypted using old long-term, encrypted keys that have been superseded. No old long-term keys are used in future transactions.
Backup and subsequent restoration of the database of a participant node supports KMS key rotation and rotation of Canton's long-term keys.
For high availability operation, Canton supports duplication of keys.

Note: Confidentiality at runtime for the private keys is out of scope. If envelope encryption is used then we do not protect against an attacker that has privileged access to the node's system and can inspect the memory.

Protect Private Keys With Envelope Encryption and a Key Management Service

Canton can protect private keys by forcing them to be internally stored in an encrypted form so they can't be decrypted and used to create fraudulent transactions. This protection at rest safeguards against malicious adversaries that get access to a node's storage layer. Keys will only be decrypted when being used and stored in a memory cache for fast access. We currently make use of a KMS's ability to securely perform this encryption/decryption of keys using a symmetric encryption key, which we call *KMS wrapper key*, without ever exposing it to the outside world, as it is backed by Hardware Security Modules (HSM) that move the crypto operations to a secure enclave.



Directly encrypting the Canton private keys with a KMS wrapper key, i.e. *envelope encryption*, has multiple advantages compared to storing these keys in the KMS itself:

Reduces the impact on performance due to additional latency and the probability of throttling KMS API requests if the thresholds are exceeded.
Preserves Canton's current key schemes, which remain flexible and easily modifiable. Not all KMS implementations offer modern and fast signing schemes such as Ed25519.

The confidentiality of the Canton private long-term and permanent keys depends on the access to the KMS wrapper key. The KMS must be locked down appropriately:

Export of the symmetric key must be forbidden.
Only authorized Canton nodes can use the wrapper key for decryption.
Usage of the wrapper key must be logged for auditability.
Separation of duties between the KMS operator and the database operator.

Externalize Private Keys With a Key Management Service

Canton can also protect private keys by outsourcing their generation and storage to a KMS, making use of of its API to perform necessary crypto operations such as decryption and signing. This protection safeguards against malicious adversaries that, besides access to the storage layer, can also access the node's system and inspect its memory. Using a KMS's underlying monitoring framework (e.g. AWS CloudTrail Logs or GCP Cloud Audit Logs) in combination with Canton logging also offers a reliable way to maintain the security, reliability of Canton, and identify any possible misuse of its private keys.

This improvement in security comes with drawbacks, in particular:

Added latency resulting from the need to use a KMS to decrypt and sign messages.
Canton's supported schemes must match those provided by the KMS.

KMS Integration

Canton currently makes use of AWS or GCP KMSs to protect its private keys. The AWS KMS API or the GCP KMS API are similar to a hardware security module (HSM) where cryptographic operations can be done within the KMS using the stored keys, without exposing them outside of the KMS.

Besides offering a secure platform to create, manage, and control cryptographic keys, they also support:

Enforcement of key usage/authorization policies;
Access to the key usage authorization logs;
Multi-region keys that allow for the replication of keys in multiple regions for disaster recovery;
Automatic rotation of keys. Note that both AWS and GCP transparently select the appropriate KMS key to use, so they can be safely rotated without any code changes.

KMS Wrapper Key Rotation

AWS and GCP KMS offer two different ways to rotate keys, either automatically or manually. By default, every symmetric key created by these KMSs is set for automatic rotation (yearly for AWS, and user-defined for GPC) where only the key material is changed. The properties of the KMS key do not change and there is no need to re-encrypt the data with the newly rotated key. Management of different key versions is done seamlessly and no changes are necessary in Canton. We recommend the *rotation of the underlying Canton long-term keys* after the KMS key has been rotated. The rotation frequency is fixed and cannot be changed.

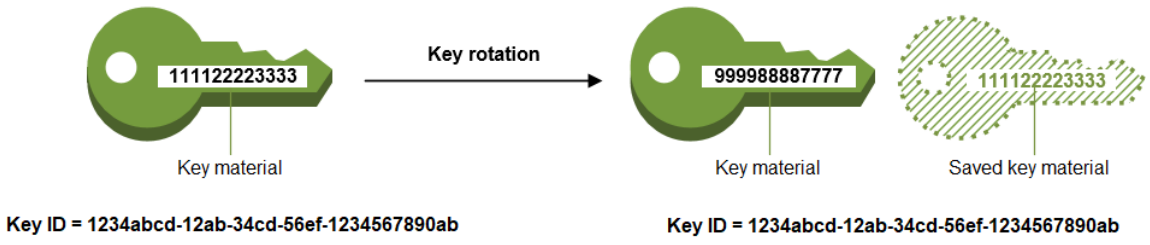


Fig. 33: image taken from https://docs.aws.amazon.com/kms/latest/developerguide/rotate-keys.html

The manual rotation of a wrapper key requires not only the creation of a new KMS key but also the re-encryption of our data with it. To do this Canton node administrators can request a *manual rotation of the KMS wrapper key through the Canton console*.

KMS Key Rotation

When Canton's signing and encryption keys are off-sourced to a KMS (rather than encrypted at rest with a KMS wrapper key) their rotation has to be operated manually. Neither AWS or GCP provide automatic asymmetric key rotation. Manual key rotation is achieved by requesting either: (1) a *standard rotation of Canton's keys*, which in this particular case also involves the rotation of the underlying KMS key, or (2) a *rotation to a previously generate KMS key*.

Satisfied Requirements

Our solutions: (a) *private key storage protection using envelope encryption* and (b) *private key externalization* comply with all the previously mentioned *requirements* in the following ways:

The long-term keys must not be available on disk or in storage in a way that would allow someone w

- The long-term and permanent keys are either: (a) only stored in an encrypted form in the database (the corresponding encryption key is stored securely by the KMS in an HSM), or (b) not stored at all by Canton.

The keys must not be part of Canton's container images.

- The Canton private keys are stored in the (a) database of the node or directly in the (b) external KMS and not in the container image. Credentials to access the KMS can be passed in via the environment when a container is created, the credentials must not be stored in the image.

A key administrator can both rotate the KMS key or long-term keys in Canton.

- Canton already supports manual rotation of long-term keys. In scenario (b) this also involves the re-generation of the keys in the KMS.
- Support of KMS wrapper key rotation (b) based on either: an KMS automated annual key rotation, or a manual rotation and re-encryption of the Canton private keys.

Historical contract data can be decrypted using old long-term, encrypted keys that have been super

- Canton already supports rotation of long-term keys with a synchronized state on which keys are active across nodes as part of topology management.

Backup and subsequent restoration of the database of a participant node supports KMS key rotatio

- Database restoration/backup is only needed for (a) protection of keys at rest and as long as the *database and the wrapper key are available*, backup and restoration are not impacted by key rotation. Replicating a KMS key in multiple regions can also mitigate the impact of a failure in the primary region.
- A KMS operator must ensure its configured key store has in place a robust disaster recovery plan to prevent the permanent loss of keys.

For high availability operation, Canton supports duplication of keys.

- Canton supports AWS and GCP multi-region keys when enabled in the configuration, as well as when the operator manually creates the key and just configures the existing key id in Canton. *Note: replicating keys to other regions is a manual process by the operator and not done automatically by Canton.*

Resilience to Malicious Participants

The Canton architecture implements the Daml Ledger Model, which has the following properties to ensure ledger integrity:

Model conformance;
Signatory and controller authorization; and
Daml ledger consensus and consistency, which contributes the most to the resilience.

An overview is presented here for how the Canton run-time is resilient to a malicious participant with these properties.

The ledger API have been designed and tested to be resilient against a malicious application sending requests to a Canton participant node. The focus here is on resilience to a malicious participant.

Model Conformance

During interpretation, the Daml engine verifies that a given action for a set of Daml packages is one of the allowed actions by the party for a contract (i.e., it conforms to the model). For example in an IOU model, it is valid that the actor of a transfer action must be the same as the owner of the contract and invalid for a non–owner to attempt a transfer, because the IOU must only be transferred by the owner.

Signatory and Controller Authorization

During interpretation, the Daml engine verifies the authorization of ledger actions based on the signatories and actors specified in the model when compared with the party authorization in the submitter information of the command.

Daml Ledger Integrity

Canton architecture ensures the integrity of the ledger for honest participants despite the presence of malicious participants. The key ingredients to achieving integrity are the following:

Deterministic transaction validation to reach consensus;
Consistent transaction ordering and validation;
Consistency checks with at least one honest participant per signatory party; and
Using an authenticated data structure (generalized blinded Merkle tree) for transactions that balances consensus with privacy.

Deterministic Transaction Execution

The execution of Daml is deterministic even though there are multiple, distributed participant nodes: given a set of Daml packages that are identified by their content and a command (create or exercise), the result of a (sub-)transaction will always be the same for the involved participant nodes. This property is used by Canton to reach agreement on whether a submitted (sub-)transaction is valid or invalid – the agreement is a requirement for ledger integrity.

Consistent Transaction Ordering and Validation

Canton uses distributed conflict detection among the involved participant nodes to ensure integrity since, by design, there is no centralized component that knows the activeness of all contracts. Instead all involved participants process the transactions in the same order so that if two concurrent transactions consume the same contract only the first transaction consumes the contract and the other transaction fails (e.g., no double spend). This means that a failed consistency check does not necessarily mean the submitter was malicious; it may be the result of a race condition in the application to consume the same contract. The sequencer node guarantees that all messages are totally ordered timestamps.

The deterministic order is established with unique timestamps from the sequencer, which implements a guaranteed total order multicast; that is, the sequencer guarantees the delivery of an end-to-end encrypted message to all all recipients. The deterministic order of message delivery results in a deterministic order of execution which ensures ledger integrity.

For finality and bounded decision times of transactions, the sequencer is immutable and append-only. In the event of a timeout, the timeouts of transactions are consistently derived from the sequencer timestamps so that timeouts are deterministic as well.

The set of recipients on the sequencer message can be validated by a recipient to ensure that the other participants of the transaction have been informed as well (i.e., guaranteed communication). Otherwise the malicious submitter would break consensus, resulting in a loss of ledger integrity where participants hosting a signatory are not informed about a state change.

Consistency With at Least One Honest Participant per Signatory Party

Although participants can verify model conformance and authorization on their own as described in the previous sections, the consistency check needs at least one honest participant hosting a signatory party to ensure consistency. If all signatories of a contract are hosted by dishonest participants, a transaction may use a contract even when the contract is not active.

Authenticated Data Structure for Transactions

The hierarchical transactions are represented by an authenticated data structure in the form of a generalized blinded Merkle tree (see https://www.canton.io/publications/iw2020.pdf). At a high level, the Merkle tree can be thought of like a blockchain in a tree format rather than a list. The Merkle tree is used to reach consensus on the hierarchical data structure while the blinding provides sub-transaction privacy. The mediator sees the shape of the transaction tree and who is involved, but no transaction payload. The entire transaction and Merkle tree is identified by its root hash. A recipient can verify the inclusion of an unblinded view by its hash in the tree. The mediator receives confirmations of a transaction for each view hash and aggregates the confirmations for the entire Merkle tree. Each participant can see all the hashes in the Merkle tree. If two participants have different hashes for the same node, the mediator will detect this and reject the transaction. The mediator also sees the number of participants involved so it can detect a missing or additional participant. The authenticated data structure ensures that participants process the same transaction and reach consensus.

Detection of Malicious Participants

In addition to the steps outlined above, the system has multiple approaches to detect malicious behavior and to keep evidence for further investigation:

Pairs of participants periodically exchange a commitment of the active contract set (ACS) for their mutual counterparties. This ensures that any diverging views between honest participants will be detected within the ACS commitment periods and participants can repair their mutual state.
Non-repudiation in the form of digital signatures enables honest participants to prove that they were honest and who was dishonest by preserving the signed responses of each participant.

Consensus & Transparency

Consensus and *Transparency* are high-level requirements that ensure that stakeholders are notified about changes to their projection of the virtual shared ledger and that they come to the same conclusions, in order to stay synchronized with their counterparties.

Operating on the Same Transaction

The Canton protocol includes the following steps to ensure that the mediator and participants can verify that they have obtained the same transaction tree given by its root hash:

(1) Every participant gets a partially blinded Merkle tree, defining the locations of the views they are privy to.
(2) That Merkle tree has a root. That root has a hash. That's the root hash.
(3) The mediator receives a partially blinded Merkle tree, with the same hash.
(4) The submitting participant will send an additional root hash message in the same batch for each receiving participant. That message will contain the same hash, with recipients being both the participant and the mediator.
(5) The mediator will check that all participants mentioned in the tree received a root hash message and that all hashes are equal.
(6) The mediator sends out the result message that includes the verdict and root hash.

An important aspect of this process is that transaction metadata, such as a root hash message, is not end-to-end encrypted, unlike transaction payloads which are always encrypted. The exact same message is delivered to all recipients. In the case of the root hash message, both the participant and the mediator who are recipients of the message get the exact same message delivered and can verify that both are the recipient of the message.

Stakeholders Are Notified About Their Views

Imagine the following attack scenarios on the transaction protocol at the point where a dishonest submitter prepares views.

Scenario 1: Invalid View Common Data

The submitter should send a view V2 to Alice and Bob (because it concerns them both as they are signatories), but the dishonest submitter tells the mediator that view V2 only requires the approval of Bob, and only sends it to Bob's participant. In this scenario both participants of Alice and Bob are honest.

Mitigation

The view common data is incorrect, because Alice is missing as an informee for the view V2. Given that Bob's participant is honest, he will reject the view by sending a reject to the mediator in the case of a signatory confirmation policy and not commit the invalid view to his ledger as part of phase 7. The two honest participants Alice and Bob thereby do not commit this invalid view to their ledger.

Scenario 2: Missing Sequencer Message Recipient

The dishonest submitter prepares a correct view common data with Alice and Bob as informees, but the corresponding sequencer message for the view is only addressed to Bob's participant. The confirmation policy does not require a confirmation from Alice's participant, e.g., VIP confirmation policy. In this scenario both participants of Alice and Bob are honest.

Mitigation

The mitigation relies on the following two properties of the sequencer:

(1) The trust assumption is that the sequencer is honest and actually delivers a message to all designated recipients (2) A recipient learns the identities of recipients on a particular message from a batch if it is itself a recipient of that message

The Bob participant can decrypt the view and verify the stakeholders against the set of recipients on the sequencer message. The mapping between parties and participants is part of the topology state on the domain and therefore the resolution is deterministic across all nodes. Seeing that the Alice participant is not a recipient despite Alice being a signatory on the view, Bob's participant will reject the view if it is a VIP participant; in any case, it will not commit the view as part of phase 7. The two honest participants Alice and Bob thereby do not commit this invalid view to their ledger.

Scenario 3: All Other Participants Dishonest

It is not required that the other participants besides Alice are honest. Let's consider a variation of the previous scenario where both the submitter and Bob are dishonest. Again Alice's participant node is not a recipient of a view message, although she is hosting a signatory. That means the view is not committed to the ledger of the honest participant Alice, because she has never seen it. Bob's participant is dishonest and approves and commits the view, although it is malformed. However, the Canton protocol does not provide any guarantees on the ledger of dishonest participants.

Scenario 4: Invalid Encryption of View

A view is encrypted with a symmetric key and the secret to derive the symmetric key for a view is encrypted for each recipient of the view with their public encryption key. The dishonest submitter produces a correct view and a complete recipient list of the corresponding sequencer message, but encrypts the symmetric key secret for Alice with an invalid key. Alice's participant will be notified about the view but unable to decrypt it.

Mitigation

If the Alice participant is a confirmer of the invalid encrypted view, which is the default confirmation policy for signatories, then she will reject the view because it is malformed and cannot be decrypted by her.

Currently the check by the other honest participant nodes that the symmetric key secret is actually encrypted with the public keys of the other recipients is missing and a documented limitation. We need to use a deterministic encryption scheme to make the encryption verifiable, which is currently not implemented.

1.44.2.7 System Architecture FAQ

What does the Sequencer do?

The sequencer nodes, together with their shared sequencer backend (blockchain or database) and the schema of the sequencer backend (native smart contracts or database schema), provide message delivery between Canton nodes that is guaranteed to be order consistent, delivery consistent and multi-cast.

Multi-cast means that Alice can send a single message to multiple recipients (Carol, Dave, etc.) as one operation.

Delivery consistent means that if Alice sends a message to Carol and Dave, then either the message gets delivered to both recipients, or neither.

Order consistent means that if Alice sends a message to Carol, Dave, and others, and Bob sends a message to Carol, Dave, then Carol and Dave see the messages from Alice and Bob in the same order.

Further Reading:

> Requirements on Sequencer Domain Entity
> Domain internal components

How does Canton process a transaction?

Canton's execution model is that the submitting participant node computes the entire transaction using the Daml interpreter and then decomposes it into views (also known as *projections*) to other participants, and then submits those views as part of a confirmation request to the other involved participants and the mediator. The participants validate the received transaction views by re-computing them with the deterministic Daml interpreter, and then send confirmations to the mediator. As all the participants received the data in the same order, the outcome is deterministic, allowing to pin-point malicious behaviour. The mediator processes the confirmations and sends out an aggregated commit message to all involved participants once sufficient confirmations are received. All messages are sent via the sequencer.

Further Reading:

Transaction processing in Canton
Daml's Execution Model
Projections

How does Canton ensure privacy?

Most sequencer backend options have limited privacy features. To provide privacy even against the operator of the sequencers and sequencer backend, Canton encrypts all message payloads sent via the sequencer to be readable only by the intended recipients. That includes the transaction payloads sent as part of confirmation requests.

Canton messages are multi-cast, meaning they can have multiple recipients, and in some cases (e.g. commit requests) have different views for the different recipients. The submitter of a message generates a single-use symmetric View Encryption Key for each view, and encrypts the views using those keys. It then encrypts only a seed for that View Encryption Key using the public half of an asymmetric Participant Encryption Keys that each Canton node publishes.

The View Encryption Keys are kept - encrypted for each receiver - with the message payload itself. A receiving node uses their Participant Encryption key to decrypt the seed of the View Encryption Key for each of the views they are entitled to read, and uses a key derivation function (HKDF to be precise) to recover the View Encryption Key and read the view.

The supported encryption algorithms for asymmetric encryption (Participant Encryption Keys) and symmetric encryption (View Encryption Keys) are listed in the documentation here.

Further Reading:

Encryption Keys
View Encryption Keys
Cryptographic Key Usage

Where does “the golden source” of Daml Ledger data live in Canton?

The short answer is that Daml Ledger data lives both on the Canton participant nodes and on the sequencer backend, meaning the blockchain or database enabled by the driver. The data is stored in the two places in different ways, but remains fully consistent thanks to Canton’s deterministic execution model.

All communication between Canton nodes, including the confirmation requests for transactions and the resulting confirmations and rejections, are stored on the sequencer backend. Since Daml and Canton are built around deterministic execution, you can thus consider that data on the sequencer backend, together with the Participant Encryption Keys, to be a complete copy of the Daml Ledger.

On the flip side, each Participant node stores its view of the Daml Ledger in an unencrypted format suitable for serving the Ledger API. The set of all participant nodes jointly holds the entire ledger state and history.

Further Reading:

Transaction processing in Canton

How is Canton able to recover from data loss?

As discussed in *Where does the golden source of Daml Ledger data live in Canton?*, the ledger data lives in two places, once encrypted in the sequencer backend, and once unencrypted spread between participant nodes. As long as you have a complete copy in either place, you can recover and continue operation.

As long as the entire ledger history in the sequencer backend is available, and you hold a participant’s Participant Encryption Keys, it is possible to recover the participant from the underlying sequencer backend. So if you use a blockchain as the sequencer backend, and can ensure that that blockchain stays available and uncorrupted, you can always recover from participant data loss.

Should your sequencer backend go down, but all participants are still up and running, you can continue running the system by coordinating all participant nodes to migrate active contracts to a new domain, with a new sequencer backend.

Should the sequencer backend no longer have the full ledger history, for example due to a domain switch, or because of deliberate Ledger Pruning, participants can still recover from a combination of the partial sequencer backend and a state snapshot. Such a snapshot can come either from a backup, or from the participants’ peers. At the time of writing this process is not fully automated but possible through Canton’s repair endpoints.

To be able to get snapshots from peers securely, nodes regularly exchange commitments via the underlying sequencer backend. You can think of these as hashes of shared state. If Alice and Bob each run a participant, Alice’s participant will regularly communicate a hash of the state it shares with Bob’s participant and vice versa. As the state is the same, the hash will be the same. This provides real-time consistency checks, allows participants to detect faulty behaviour in domain components, and also helps recovery in the above scenario. Alice can ask Bob for a snapshot of her data shared with Bob, and check its correctness by comparing it to the commitment she made on the sequencer backend.

Further Reading:

Repairing Participants
Backup and Restore

Ledger Pruning

1.44.3 Frequently Asked Questions

This section covers other questions that frequently arise when using Canton. If your question is not answered here, consider searching the Daml forum and creating a post if you can't find the answer.

1.44.3.1 Log Messages

Database task queue full

If you see the log message:

```
java.util.concurrent.RejectedExecutionException:
Task slick.basic.BasicBackend$DatabaseDef$@... rejected from slick.util.
↪AsyncExecutorWithMetrics$$...
[Running, pool size = 25, active threads = 25, queued tasks = 1000, completed
↪tasks = 181375]
```

It is likely that the database task queue is full. You can check this by inspecting the log message: if the logged `queued tasks` is equal to the limit for the database task queue, then the task queue is full. This error message does not indicate that anything is broken, and the task will be retried after a delay.

If the error occurs frequently, consider increasing the size of the task queue:

```
canton.participants.participant1.storage.config.queueSize = 10000
```

A higher queue size can lead to better performance, because it avoids the overhead of retrying tasks; on the flip side, a higher queue size comes with higher memory usage.

Serialization Exception

In some situations, you might observe the following log message in your log file or in the database log file:

```
2022-08-18 09:32:39,150 [] INFO  c.d.c.r.DbStorageSingle - Detected an
↪SQLException. SQL state: 40001, error code: 0
    org.postgresql.util.PSQLException: ERROR: could not serialize access due to
↪concurrent update
```

This message is normally harmless and indicates that two concurrent queries tried to update a database row and due to the isolation level used, one of them failed. Currently, there are a few places where we use such queries. The Postgres manual will tell you that an application should just retry this query. This is what Canton does.

Canton's general strategy with database errors is to retry retryable errors until the query succeeds. If the retry does not succeed within a few seconds, a warning is emitted, but the query is still retried.

This means that even if you turn off the database under full load for several hours, under normal circumstances Canton will immediately recover once database access has been restored. There is

no reason to be concerned functionally with respect to this message. As long as the message is logged on INFO level, everything is running fine.

However, if the message starts to appear often in your log files, you might want to check the database query latencies, number of database connections and the database load, as this might indicate an overloaded database.

1.44.3.2 Console Commands

I received an error saying that the DomainAlias I used was too long. Where I can see the limits of String types in Canton?

Generally speaking, you don't need to worry about too-long Strings as Canton will exit in a safe manner, and return an error message specifying the String you gave, its length and the maximum length allowed in the context the error occurred. Nonetheless, the known subclasses of LengthLimitedStringWrapper and the type aliases defined in the companion object of LengthLimitedString list the limits of String types in Canton.

1.44.3.3 Bootstrap Scripts

Why do you have an additional new line between each line in your example scripts?

When we write `participant1 start` the scala compiler translates this into `participant1.start()`. This works great in the console when each line is parsed independently. However with a script all of its content is parsed at once, and in which case if there is anything on the line following `participant1 start` it will assume it is an argument for `start` and fail. An additional newline prevents this. Adding parenthesis would also work.

How can I use nested import statements to split my script into multiple files?

Ammonite supports splitting scripts into several files using two mechanisms. The old one is `interp.load.module(..)`. The new one is `import $file.<fname>`. The former will compile the module as a whole, which means that variables defined in one module cannot be used in another one as they are not available during compilation. The `import $file.` syntax however will make all variables accessible in the importing script. However, it only works with relative paths as e.g. `../path/to/foo/bar.sc` needs to be converted into `import $file.^.path.to.foo.bar` and it only works if the script file is named with suffix `.sc`.

How do I write data to a file and how do I read it back?

Canton uses Protobuf for serialization and as a result, you can leverage Protobuf to write objects to a file. Here is a basic example:

```
// Obtain the last event.
val lastEvent: PossiblyIgnoredProtocolEvent =
  participant1.testing.state_inspection
    .findMessage(da.name, LatestUpto(CantonTimestamp.MaxValue))
    .getOrElse(throw new NoSuchElementException("Unable to find last event."))
```

(continues on next page)

(continued from previous page)

```
// Dump the last event to a file.
utils.write_to_file(lastEvent.toProtoV0, dumpFilePath)

// Read the last event back from the file.
val dumpedLastEventP: v0.PossiblyIgnoredSequencedEvent =
  utils.read_first_message_from_file[v0.PossiblyIgnoredSequencedEvent](
    dumpFilePath
  )

val dumpedLastEventOrErr: Either[
  ProtoDeserializationError,
  PossiblyIgnoredProtocolEvent,
] =
  PossiblyIgnoredSequencedEvent
    .fromProtoV0(testedProtocolVersion, cryptoPureApi(participant1.config))(
      dumpedLastEventP
    )
```

You can also dump several objects to the same file:

```
// Obtain all events.
val allEvents: Seq[PossiblyIgnoredProtocolEvent] =
  participant1.testing.state_inspection.findMessages(da.name, None, None, None)

// Dump all events to a file.
utils.write_to_file(allEvents.map(_.toProtoV0), dumpFilePath)

// Read the dumped events back from the file.
val dumpedEventsP: Seq[v0.PossiblyIgnoredSequencedEvent] =
  utils.read_all_messages_from_file[v0.PossiblyIgnoredSequencedEvent](
    dumpFilePath
  )

val dumpedEventsOrErr: Seq[Either[
  ProtoDeserializationError,
  PossiblyIgnoredProtocolEvent,
]] =
  dumpedEventsP.map {
    PossiblyIgnoredSequencedEvent.fromProtoV0(
      testedProtocolVersion,
      cryptoPureApi(participant1.config),
    )(_)
  }
```

Some classes do not have a (public) `toProto*` method, but they can be serialized to a ByteString instead. You can dump the corresponding instances as follows:

```
// Obtain the last acs commitment.
val lastCommitment: AcsCommitment = participant1.commitments
  .received(
    da.name,
    CantonTimestamp.MinValue.toInstant,
    CantonTimestamp.MaxValue.toInstant,
  )
```

(continues on next page)

(continued from previous page)

```
  .lastOption
  .getOrElse(
    throw new NoSuchElementException("Unable to find an acs commitment.")
  )
  .message

// Dump the commitment to a file.
utils.write_to_file(
  lastCommitment.toByteString,
  dumpFilePath,
)

// Read the dumped commitment back from the file.
val dumpedLastCommitmentBytes: ByteString =
  utils.read_byte_string_from_file(dumpFilePath)

val dumpedLastCommitmentOrErr: Either[
  ProtoDeserializationError,
  AcsCommitment,
] =
  AcsCommitment.fromByteString(dumpedLastCommitmentBytes)
```

1.44.3.4 Why is Canton complaining about my database version?

Postgres

Canton is tested with Postgres 10, 11, 12, 13, and 14 - so these are the recommended versions. Canton is also likely to work with any higher versions, but will WARN when a higher version is encountered. By default, Canton will not start when the Postgres version is below 10.

Oracle

Canton Enterprise additionally supports using Oracle for storage. Only Oracle 19 has been tested, so by default Canton will not start if the Oracle version is not 19.

Note that Canton's version checks use the `v$$version` table so, for the version check to succeed, this table must exist and the database user must have `SELECT` privileges on the table.

Using non-standard database versions

Canton's database version checks can be disabled with the following config option:

```
canton.parameters.non-standard-config = yes
```

Note that this will disable all standard config checks, not just those for the database.

1.44.3.5 How do I enable unsupported features?

Some alpha / beta features require you to explicitly enable them. However, please note that none of them are supported by us in our commercial product and that turning them on will very much likely break your system:

```
canton.parameters {
  # turn on non-standard configuration support
  non-standard-config = yes

  # turn on support of development version support for domain nodes
  dev-version-support = yes
}

canton.domains.mydomain.init.domain-parameters {
  # set the domain protocol version to `dev` (or to any other unstable protocol
↪version)
  # requires you to explicitly enable non-standard-config. not to be used for
↪production.
  protocol-version = dev
}

canton.participants.participant1.parameters = {
  # enable dev version on the participant (this will allow the participant to
↪connect to a domain with dev protocol version)
  # and it will turn on support for unsafe daml lf dev versions
  # not to be used in production and requires you to define non-standard-config =
↪yes
  dev-version-support = yes
}
```

1.44.3.6 How to troubleshoot included configuration files?

If Canton is unable to find included configuration files, please read the section on *including configuration files* and the HOCON specification. Additionally, you may run Canton with `-Dconfig.trace=loads` to get trace information when the configuration is parsed.

1.45 Test Evidence

Daml is publishing test evidence for the most important traits of tests: Security, Operability, Functional and Reliability.

It can be found in the relevant releases page, under *Assets*.

1.46 Participant Query Store User Guide

1.46.1 Introduction

The term operational data store (ODS) usually refers to a database that mirrors the ledger and allows for efficient querying. The Participant Query Store (PQS) feature acts as an ODS for the participant node. It stores contract creation, contract archival, and exercise information in a PostgreSQL database using a JSONB column format. You access the data using SQL over a JDBC connection.

The PQS is intended for high throughput and complex queries, for which the Canton ledger (gRPC Ledger API) and the JSON API are not optimized. The PQS is useful for:

Application developers to access data on the ledger, observe the evolution of data, and debug their applications.
Business analysts to analyze ledger data and create reports.
Support teams to debug any problems that happen in production.
Application operators to take a full snapshot of the ledger (from the start of the ledger to the current latest transaction). Alternatively, they can take a partial snapshot of the ledger (between specific offsets).
Report writers to extract historical data and then stream indefinitely (either from the start of the ledger or from a specific offset).

There are many other uses.

In the Early Access implementation, the PQS provides a unidirectional path for exporting data from the ledger event stream to a PostgreSQL data store. Data is exported in an append-only fashion and provides a stable view of data for purposes such as point-in-time queries.

1.46.2 Early Access Purpose and Limitations

The Early Access (EA) release allows users to learn about the architecture and programming model of the PQS. This enhancement to the participant node provides new capabilities that can take time to explore. The EA release is not fully production ready, but it will rapidly become enterprise-hardened. Since applications take time to develop, the EA version is recommended for learning and development. As enhancements are made and gaps are closed, a new patch release will be provided.

The current limitations of the EA version are:

PQS has not been performance optimized, so it is not yet ready for large- or high-throughput queries.
As is typical of EA releases, backward compatibility between EA releases may be sacrificed to make improvements in the user experience and design of PQS. You may need to make adjustments in your use through the EA period.

Future early access releases will remove or reduce these limitations. Please check back to this section for announcements of a new early access release.

1.46.2.1 Early Access Release Versions

The historical table below lists the available Early Access releases of the Participant Query Store. Click the date to download the JAR.

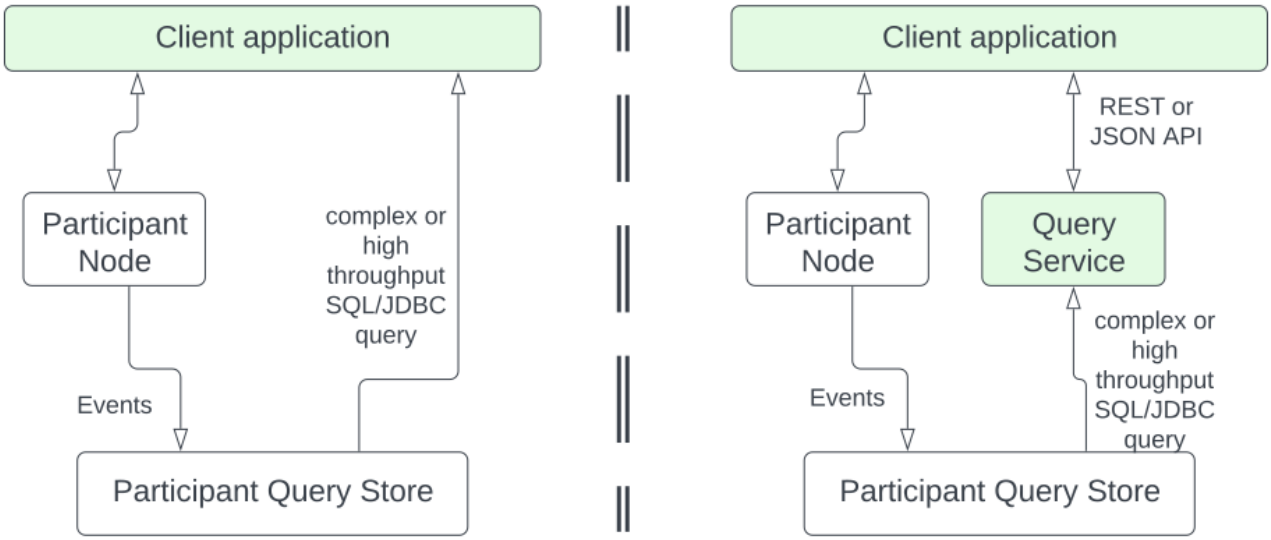
Date	Description
2023-08-09	Initial early access release.
2023-08-31	Added OAuth support.
2023-09-06	Documentation updated. Added *PQS Schema Design*, *Offset Management*, *Querying Patterns*, *Advanced Querying Topics* sections.
2023-09-18	Documentation updated. Updated command line options and added information about using `--pipeline-filter` option.
2023-09-19	New release. JDBC driver fix to not inject `?`. `--target-postgres-autoapplyschema` renamed to `--target-schema-autoapply`
2023-09-22	New release. Added pruning documentation. Environment variables now have `SCRIBE_` prefix to avoid name clashes. Updated the `--pipeline-parties` option information.
2023-09-26	New release. The filter is now applied on the DB functions, such as choices.
2023-10-06	New release. Fix a JWT audience bug. Name format change.

1.46.3 Overview

1.46.3.1 Architecture

The typical configuration is to have a separate PQS instance with its own DB for each participant node (shown in the figure). In this configuration, the PQS extracts contract information for all parties on the participant node. As data is physically segregated by Daml participants and hosted Daml parties must trust the node operator, they may trust the operator to protect the PQS privacy as well. A more restricted configuration is possible that limits the parties for which the PQS extracts information.

A client application can access the PQS directly using a JDBC connection where the data access rights are defined by the PostgreSQL database (left side in the figure). In this case, having access to the database means that the user has access to all the content in the database. If finer-grained access is needed, a read-only query service (right side in the figure) can be inserted between the client application and the PQS. That query service can filter out what a client application can see. This is a fairly standard pattern in the industry.



To understand the format that PQS outputs into a Postgres document-oriented cache, you must understand how the ledger stores data. The Daml ledger is composed of transactions, which generate events. An event can represent one of these situations:

Creation of contracts (create event)
Exercise of a choice on a contract (exercise event), which archives the contract if it is a consuming choice

A contract on the ledger is either created or archived. The relationships between transactions and contracts are captured in the database as follows:

All contracts have links (foreign keys) to the transaction in which they were created.
Archived contracts have pointers to the transaction in which they were archived.

Transactions on the ledger are inserted into PostgreSQL concurrently, for high performance. Consistency (for readers) is provided through a watermark mechanism that indicates a consistent offset from which readers can consume for a fully consistent ledger. These details are managed for readers through the functions available in PostgreSQL. Depending on your needs, readers may wish to use or bypass these mechanisms, depending on the type of query and consistency required.

1.46.3.2 PQS Schema Design

PQS is not directly involved in querying/reading the datastore - the application is free to query it, such as via JDBC. The objectives of the schema design is to facilitate:

Scaleable writes: transactions are written in parallel, so writes do not need to be sequential.
Scaleable reads: queries can be parallelized and are not blocked by writes. They produce sensible query plans with no unnecessary table scans.
Ease of use: readers are able to use familiar tools and techniques to query the datastore, without needing to understand the specifics of the schema design. Simple entry points provide access to data in familiar ways. In particular, readers do not need to navigate the offset-based model.
Read consistency: readers are able to achieve the level of consistency that they require, including consistency with other ledger datastores, or with ledger commands that have been executed.

The following principles apply:

Append-only: only INSERTs are used, and no UPDATEs or DELETEs are used in transaction processin.
Offset-based: all physical tables are indexed by offset, meaning that all ledger data is known in terms of the offset in which it was committed to the ledger.
Implicit offset: readers can opt for queries with implicit offset, meaning they can ignore the role of offset in their queries - but still provide a stable view of the ledger data. Much like PostgreSQL provides MVCC capabilities without the reader needing to understand the underlying implementation, we seek to provide a similar experience for readers of the ledger data.
Idempotent: PQS is designed to be restarted at any time, and will not impact the integrity of the data. This is achieved by using the offset-based model and ensuring that (other than the datastore itself) PQS is stateless.
Watermarks: PQS maintains a watermark of the latest contigous offset, representing the point of the ledger that has been fully processed. This is used to ensure that the ledger data has read consistency, without needing readers to perform pathalogical table scans to achieve this. This resolves the uncertainty created by the parallel writes.

1.46.3.3 JSON Data

Relational databases excel at storing structured data for which the schema is known in advance. However, they have traditionally lacked mechanisms for data that is more dynamic or evolves. For example, you may want to store arbitrary Daml contracts and might prefer not to update the database schema every time the underlying template changes.

PostgreSQL helps manage unstructured data through native support for JSON data and allows queries to process this data. For best performance, the PQS stores data as JSONB only.

An example query might look like this:

```
SELECT *
FROM contract
WHERE payload->>'isin' = 'abc123'
ORDER BY payload->'issuanceData'->'issueDate'->>'Some';
```

For more information on querying JSON data, see the section JSON Functions and Operators in the PostgreSQL manual. The operators `->`, `->>`, `#>`, `#>>`, and `@>` may be of particular interest.

This *section below* summarizes how the ledger data is encoded in JSON.

1.46.3.4 Continuity

The PQS is intended for continuous operation. Upon restart after an interruption, PQS determines the last consistent offset and continues incremental processing from that point onward. PQS terminates when encountering any error and leaves it up to the orchestration layer (such as Kubernetes) or the operator to determine the appropriate course of action.

1.46.3.5 High Availability

Multiple isolated instances of PQS can be instantiated without any cross-dependency. This allows for an active-active high availability, clustering model. Please note that different instances might not be at the same offset due to different processing rates or other factors. After querying one active instance, it is possible for you to see results that are not yet visible on an alternative, active instance. This requires consideration for the client to handle the situation where waiting or a retry is required to service at least up to requests.

1.46.4 Installing and Starting PQS

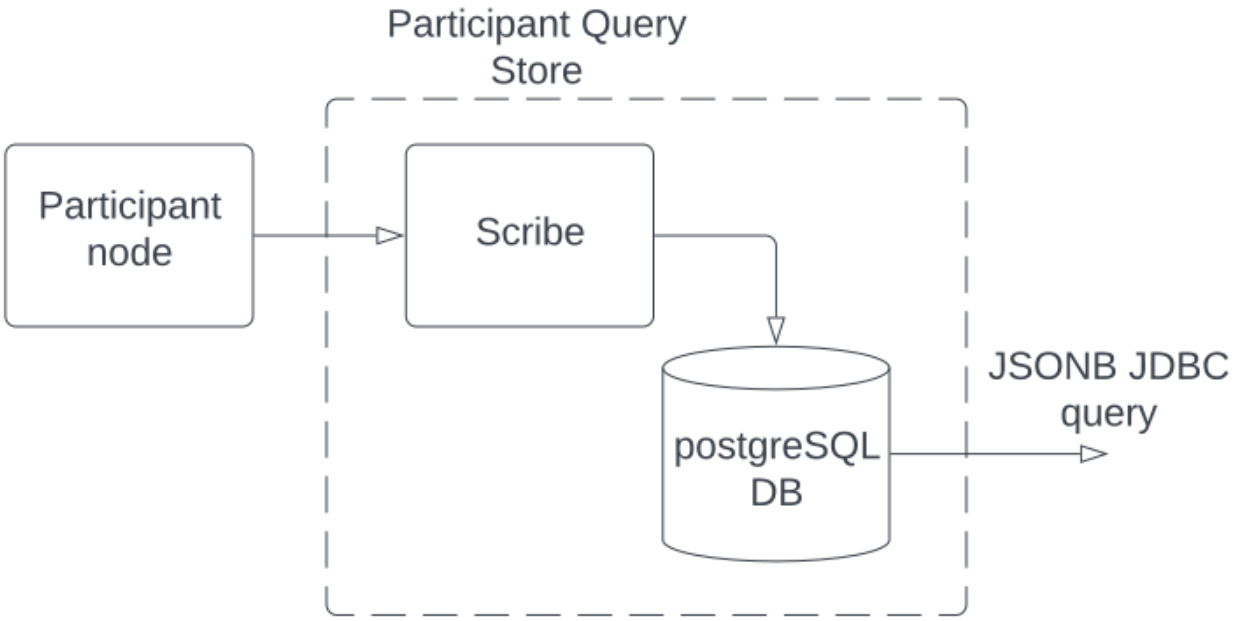
1.46.4.1 Meeting Prerequisites

Here are the prerequisites to run PQS:

A PostgreSQL database that can be reached from the PQS. Note that PQS uses the JSONB data type for storing JSON data, which requires Postgres versions 11, 13, and 15.
An empty database (recommended) to avoid schema and table collisions.
Daml ledger as the source of events. m/TLS is supported for the participant node ledger API. Alternatively, it can run against the `Sandbox`.
Installation of The Daml Enterprise SDK.

1.46.4.2 Deploying the Scribe Component

The PQS consists of two components: the PostgreSQL database and a ledger component called *Scribe*, as shown below. Scribe is packaged as a Java JAR file. To run the PQS during Early Access, retrieve `scribe.jar` from this Artifactory path.



1.46.4.3 Connecting the PQS to a Ledger

To connect to the participant node ledger, provide separate address and port parameters. For example, you could specify `--host 10.1.1.10 --port 6865`, or in short form `-h 10.1.1.168 -p 6865`.

You do not need to pass the default host `localhost` and default port `6865`.

To connect to a participant node, you might need to provide TLS certificates. To see options for this, refer to the output of the `--help` command.

1.46.4.4 Authorizing PQS

If you are running PQS against a participant node's ledger API that verifies authorization, you must provide credentials for the OAuth Client Credentials Flow. For example:

```
$ ./scribe.jar pipeline ledger postgres-document \
    --source-ledger-auth OAuth \
    --pipeline-oauth-clientid my_client_id \
    --pipeline-oauth-clientsecret deadbeef \
    --pipeline-oauth-cafile ca.crt \
    --pipeline-oauth-endpoint https://my-auth-server/token
```

The type of access token that PQS expects is Audience / Scope based tokens (see User Access Tokens for more information).

Scribe will obtain tokens from the Authorization Server on startup, and it will reauthenticate before the token expires. If Scribe fails authorization, it will terminate with an error for the service orchestration infrastructure to respond appropriately.

If you are not authenticated, there is no user to connect to a list of `readAs` parties, so you must specify the parties using the `-pipeline-parties` argument. This argument acts as a filter, restricting the data to only what's visible to the supplied list of party identifiers.

The authentication of PQS needs to match the participant nodes (PN) setup. For example, if PQS is run with authentication by setting OAuth and the PN is not configured to use authentication, then an error will result. The error will have a message like `requests with an empty user-id are only supported if there is an authenticated user`.

1.46.4.5 Setting Up PostgreSQL

To connect the database, create a PostgreSQL database with three users:

Ops: Provides a way for database administrators or Scribe to access DDL for schema creation and general maintenance.
Writer: Allows Scribe to connect, such as during pipeline operations of writing the ledger.
Reader: Supports all other users.

1.46.4.6 Connecting to the PQS PostgreSQL Data Store

The database connection is handled by the JDBC API, so you need to provide the following (all have defaults):

Hostname
Port number
Username
Password

The following example connects to a PostgreSQL instance running on localhost on the default port, with a user Postgres which has not set a password and a database called `daml_pqs`. This is a typical setup on a developer machine with a default PostgreSQL install.

```
$ ./scribe.jar pipeline ledger postgres-document \
    --target-postgres-database daml_pqs
```

The next example connects to a database on host `192.168.1.12`, listening on port `5432`. The database is called `daml_pqs`.

```
$ ./scribe.jar pipeline ledger postgres-document \
    --target-postgres-host 192.168.1.12 \
    --target-postgres-database daml_pqs
```

1.46.4.7 Logging

By default, the PQS logs to `stderr`, with `INFO` verbose level. To change the level, use the `--logger-level enum` option, as in the example `--logger-level Trace`.

1.46.4.8 Using Command Line Options

You can discover commands and parameters through the embedded `--help` (remember to include `pipeline` before `--help`), as shown in the following example.

```
./scribe.jar pipeline --help
Usage: pipeline SOURCE TARGET [OPTIONS]

Initiate continuous ledger data export

Available sources:
  ledger   Daml ledger

Available targets:
  postgres-document   Postgres database (w/ document payload representation)
  postgres-relational Postgres database (w/ relational payload representation)

Options:
  --config file                                Path to configuration overrides
↪via an external HOCON file (optional)
  --pipeline-parties string                    Daml party identifiers to filter
↪on (comma-separated) (default: List())
  --pipeline-oauth-clientid string             Client's identifier (optional)
  --pipeline-oauth-cafile file                 Trusted Certificate Authority (CA)
↪certificate (optional)
```

(continues on next page)

(continued from previous page)

```
  --pipeline-oauth-endpoint uri                Token endpoint URL (optional)
  --pipeline-oauth-clientsecret string         Client's secret (optional)
  --pipeline-filter string                     Filter expression determining
↪which templates and interfaces to include (default: *)
  --pipeline-ledger-start [enum | string]      Start offset (default: Latest)
  --pipeline-ledger-stop [enum | string]       Stop offset (default: Never)
  --pipeline-datasource enum                   Ledger API service to use as data
↪source (default: TransactionStream)
  --logger-level enum                          Log level (default: Info)
  --logger-mappings map                        Custom mappings for log levels
  --logger-format enum                         Log output format (default: Plain)
  --logger-pattern [enum | string]             Log pattern (default: Plain)
  --target-postgres-host string                Postgres host (default: localhost)
  --target-postgres-tls-mode enum              SSL mode required for Postgres
↪connectivity (default: Disable)
  --target-postgres-tls-cert file              Client's certificate (optional)
  --target-postgres-tls-key file               Client's private key (optional)
  --target-postgres-tls-cafile file            Trusted Certificate Authority (CA)
↪certificate (optional)
  --target-postgres-password string            Postgres user password (default:
↪********)
  --target-postgres-username string            Postgres user name (default:
↪postgres)
  --target-postgres-database string            Postgres database (default:
↪postgres)
  --target-postgres-port int                   Postgres port (default: 5432)
  --target-schema-autoapply boolean            Apply metadata inferred schema on
↪startup (default: true)
  --source-ledger-host string                  Ledger API host (default:
↪localhost)
  --source-ledger-auth enum                    Authorisation mode (default:
↪NoAuth)
  --source-ledger-tls-cafile file              Trusted Certificate Authority (CA)
↪certificate (optional)
  --source-ledger-tls-cert file                Client's certificate (leave empty
↪if embedded into private key file) (optional)
  --source-ledger-tls-key file                 Client's private key (leave empty
↪for server-only TLS) (optional)
  --source-ledger-port int                     Ledger API port (default: 6865)
```

For more help, use the command:

```
./scribe.jar pipeline --help-verbose
```

Following is an example of a basic command to run PQS to extract all data, including exercises, for a party with the display name Alice. You can replace the argument values with those that match your environment.

```
$ ./scribe.jar pipeline ledger postgres-document \
--pipeline-parties
 ↪Alice::12209942561b94adc057995f9ffca5a0b974953e72ba25e0eb158e05c801149639b9 \
--pipeline-datasource TransactionTreeStream \
--source-ledger-host localhost \
--source-ledger-port 6865 \
--target-postgres-host localhost \
```

(continues on next page)

(continued from previous page)

```
--target-postgres-port 5432 \
--target-postgres-database postgres \
--target-postgres-username postgres \
--target-postgres-password postgres
```

NOTE: Only `postgres-document` is currently implemented, with `postgres-relational` to follow soon.

The `-pipeline-ledger-start` argument is an enum with the following possible values:

> `Latest`: Use latest offset that is known or resume where it left off. This is the default behavior, where streaming starts at the latest known end. The first time you start, this will result in PQS calling `ActiveContractService` to get a state snapshot, which it will load into the `_creates` table. It will then start streaming creates, archives, and (optionally) exercises from the offset of that `ActiveContractService`. When you restart PQS, it will start from the point it last left off. You should always use this mode on restart.
> `Genesis`: Use the first original offset of the ledger. This causes PQS to try to start from offset `0`. It allows you to load historic creates, archives or (optionally) exercises from a ledger that already has data on it. If you try to restart on an already populated database in this mode, PQS will rewrite data if it needs to.
> `Oldest`: Use the oldest available (unpruned) offset on the ledger or resume where it left off.

PQS is able to start and finish at prescribed ledger offsets, specified by the arguments `--pipeline-ledger-start` and `--pipeline-ledger-stop`. The `./scribe.jar pipeline --help-verbose` command provides extensive help information.

The `--pipeline-filter string` option needs a filter expression to determine which templates and interfaces to include. A filter expression is a simple wildcard inclusion statement with basic boolean logic, where whitespace is ignored. Below are some examples:

> `*`: everything, which is the default
> `a.b.c.Bar`: just this one fully qualified name
> `a.b.c.*`: all under the `a.b.c` namespace
> `deadbeef..:a.b.c.Foo` just this one fully qualified name from this specific package ID
> `!a.b.c.Bar`: everything except this fully qualified name
> `a.b.c.Foo & a.b.c.Bar`: this is an error because it can't be both
> `(a.b.c.Foo | a.b.c.Bar)`: these two fully qualified names
> `(a.b.c.* & !(a.b.c.Foo | a.b.c.Bar) | g.e.f.Baz)`: everything in `a.b.c` except for `Foo` and `Bar`, and also include `g.e.f.Baz`

The `--pipeline-parties` option supports the same filter expressions as the `--pipeline-filter`. So to filter for two parties `alice::abc123...` and `bob::def567...`, you could write `--pipeline-parties="(alice* | bob*)"`. If you want to specify a specific party then include the hash behind the party hint (i.e. `Alice_1::122055fc4b190e3ff438587b699495a4b6388e911e2305f7e013af160f49a76080ab`).

Please note that the separator is a pipe character (`|`) instead of comma.

Brackets are unnecessary for simple expressions. A simple list is `--pipeline-parties="Alice_1::122055fc4b190e3ff438587b699495a4b6388e911e2305f7e013af | Alice_2::122053933e4803c2995e41faa8a29981ca0d1faf6b4ffbf917ba1edd0db133acb634 | Peter-1::358400000000000000000000000` Specifying the parties in a short form can be done by using the `*` as a wildcard. For example, `--pipeline-parties="Alice* | *358400000000000000000000000"` will select `Alice_1`, `Alice_2`, and `Peter-1`.

More advanced expressions can make use of brackets, such as `--pipeline-parties="Alice* | Bob* | (participant* & !(participant3::*))"`.

1.46.4.9 Handling Configuration Changes

PQS initializes its behavior on startup by reading its configuration files. It currently doesn't support dynamic configuration updates so making a configuration change (e.g., adding a new party, new template, or new interface) requires stopping PQS, modifying its configuration, and then starting PQS. Then, on startup, PQS will read the updated configuration.

When the configuration changes, the default is that PQS will not go back in time (older offset) but only move forward in time (current watermark offset and newer). If the database is dropped then PQS can be started at the oldest, unpruned offset of the participant node and use the participant node's history to extract the events based on the updated configuration.

1.46.5 PQS Development

1.46.5.1 Offset Management for Querying

The following functions control the temporal perspective of the ledger, considering how you wish to consider time as a scope for your queries. You may wish to:

Effectively ignore time; simply query the *latest available* state
Query the state of the ledger at a specific time in history
Query the ledger events across a time range - eg. an audit-trail
Query the ledger in a way that preserves consistency with other interactions you have had with the ledger (reader or writer)

The following functions allow you to control the temporal scope of the ledger, which establishes the context in which subsequent queries in the PostgreSQL session will execute:

`set_latest(offset)`: nominates the offset of the latest data to include in observing the ledger. If NULL then it uses the very latest available. The actual offset that will be used, is returned. If the supplied offset is beyond what is available, an error occurs.
`set_latest_minimum(offset)`: provides the minimum offset that should be used, but a more recent offset will always be chosen. Returns an error if the nominated offset is not yet available. Function returns the actual offset used.
`set_oldest(offset)`: nominates the offset of the oldest events to include in query scope. If `NULL` then it uses the oldest available. Function returns the actual offset used. If the supplied offset is beyond what is available, an error occurs.
`get_offset(time)`: a helper function to determine the offset of a given `time` (or interval prior to now).

Under this temporal scope, the following table functions allow access to the ledger and are used directly in queries. They are designed to be used in a similar manner to tables or views, and allow users to focus on the data they wish to query, with the impact of offsets removed.

`active(name)`: active instances of the target contracts/interfaces that existed at the time of the latest offset
`creates(name)`: create events that occurred between the oldest and latest offset
`archives(name)`: archive events that occurred between the oldest and latest offset
`exercises(name)`: exercise events that occurred between the oldest and latest offset

The functions allow the user to focus on the templates/interfaces/choices they wish to query, without concern for PostgreSQL name limits. The `name` parameter can be used with or without the package specified:

Fully qualified: `<package-id>:<module>:<template|interface|choice>`
Partially qualified: `<module>:<template|interface|choice>`

1.46.5.2 Querying Patterns

Several common ways to use the table functions are described next which are:

Use the most recent available state of the ledger
Query the ledger using a point in time
Query the ledger from a fixed offset
Set the oldest offset to consider
Set the oldest and latest offset by time value
Set a minimum offset for consistency
Use the widest available offset range for querying

Of course, these can be combined or altered based on the purpose of the query.

Use the Most Recent Available State of the Ledger

A user who wants to query most recent available state of the ledger. This user treats the ledger Active Contract Set as a virtual database table, and is not concerned with offsets because they want the latest result.

This user simply wants to query the (latest) state of the ledger, without consideration for offsets. Querying is inherently limited to one datasource, as the user has no control over the actual offset that will be used.

In this scenario the user wishes to query all Daml templates of `User` within the `Test.User` templates, where the user is not an administrator:

```
set_offset_latest(NULL);
SELECT *
  FROM active('Test.User:User') AS "user"
  WHERE NOT "user"."admin";
```

By using PostgreSQL's JSONB querying capabilities, we can join with the related `Alias` template to provide an overview of all users and their aliases:

```
set_latest(NULL);
SELECT "user".*, alias.*
  FROM active('Test.User:User') AS "user"
    LEFT JOIN active('Test.User:Alias') AS alias
      ON "user".payload->>'user_id' = alias.payload->>'user_id';
```

Historical events can also be accessed; by default all the history in the datastore is available for querying. The following query will return the data associated with all `User` contracts that were archived in the available history:

```
set_latest(NULL);
set_oldest(NULL);
SELECT c.*
  FROM archive('Test.User:User') AS a
    JOIN create('Test.User:User') AS c USING contract_id;
```

Query the Ledger Using a Point in Time

A report writer wants to query the ledger as of a known historical point in time, to ensure that consistent data is provided regardless of where the ledger subsequently evolved.

This user can obtain a point-in-time view of the ledger, to see all non-admin `User` templates that were active at that point in time:

```
set_latest(get_offset('2020-01-01 00:00:00+0'));
SELECT "user".*
  FROM active('Test.User:User') AS "user"
  WHERE NOT "user".admin;
```

In addition the user can then query the history of the ledger, to see how many aliases had have existed for each of these users who were active at the snapshot time

```
set_latest(get_offset('2020-01-01 00:00:00+0'));
set_oldest(NULL);
WITH "users" AS (
  SELECT  "user".*
    FROM active('Test.User:User') AS "user"
    WHERE NOT "user".admin
)
SELECT "user".user_id, COUNT(alias.*) AS alias_count
  FROM active('Test.User:User') AS "user"
    JOIN create('Test.User:Alias') AS alias
      ON "user".payload->>'user_id' = alias.payload->>'user_id'
  WHERE NOT "user".admin;
```

Query the Ledger from a Fixed Offset

An automation user who wants to query from fixed known offsets, still wants to write their query in the same familiar way.

```
-- fails if the datastore has not yet reached the given offset
set_latest("00000001250");
```

The queries will now observe active contracts from the given offset. Therefore the example queries presented above are unchanged.

Set the Oldest Offset to Consider

A user wants to present a limited amount of history to their users.

If readers wish to limit the event history, they can also call:

```
-- fails if this offset has already been pruned
set_oldest("00000000500");
```

This adjustment in scope does not affect the example queries presented above.

Set the Oldest and Latest Offset by Time Value

A user wants to present a time-based view to their users, to provide reports based on point-in-time rather than offsets

```
set_latest(get_offset(TIMESTAMP '2020-03-13 00:00:00+0'))
set_oldest(get_offset(INTERVAL '14 days')); -- history of the past 14 days
```

Set a Minimum Offset for Consistency

A website user who wants to query active contracts, after having completed a command (write) which has updated the ledger. The user does not want to see a version of the ledger prior to the command being executed.

```
-- The user just executed a command at offset #00000001350.
-- This function call will fail if the datastore has not yet reached this offset,
↪in order to provide consistent reads.
-- If it has an even more recent offset (eg. 00000001355) - this will be used
↪instead.
set_latest_minimum("00000001350");
```

Use the Widest Available Offset Range for Querying

A user wants to enquire where the datastore is up to, in terms of offset availability.

Here the user asks for the very latest and oldest offsets available to be used, and in the process returns what these offsets are:

```
SELECT set_latest(NULL) AS latest_offset, set_oldest(NULL) AS oldest_offset;
```

1.46.5.3 Advanced Querying Topics

Reading

As outlined, there are two distinct approaches used when querying ledger data in the datastore: state or events.

State, in the form of the Active Contract Set, by the function `active(name)` uses the latest offset only, using the following rules:

```
creation_offset <= latest_offset; AND
no archive_offset <= latest_offset
```

Events (create, exercise, archive) make use of the range oldest and latest offset:

```
event_offset <= latest_offset; AND
event_offset >= oldest_offset
```

Write Pipeline

Only advanced users should be concerned with the manner in which the write pipeline is implemented. The above Read API takes into consideration the manner in which the write pipeline is implemented, and therefore the above Read API is the recommended way to query the datastore. However, for completeness we provide the following information.

A Daml transaction is a collection of events that take effect on the ledger atomically. However it needs to be noted that for performance reasons these transactions are written to the datastore *in parallel*, and although the datastore is written to in a purely append-only fashion, it is not guaranteed that these transactions will become visible to readers in order. The offset-based model makes the database's isolation level irrelevant - so the loosest model (`read uncommitted`) is not harmful.

The first thing to consider when querying the datastore is the type of read consistency required. If there is no need for consistency (eg. reading a historical contract - regardless of lifetime) then payload tables can be queried directly, without any consideration of offset. Another example is a liveness metric query that calculates the number of transactions in the datastore over the past minute. Again, this could be entirely valid without consideration of the parallel-writing method.

When consistency is required, the reader must be aware of the offset from which they are reading. This ensures they do not also read further offsets that are present, but their precedent events are not yet stored in the database.

To achieve the level of consistency that you require, including read-consistency with other ledger data or commands you have executed. This is achieved by providing a function that returns the latest checkpoint offset:

```
-- utility functions
create or replace function latest_checkpoint()
returns table ("offset" _transactions."offset"%type, ix _transactions.ix%type) as
↪$$
  select max(groups."offset") as "offset", max(groups."ix") as ix
  from (SELECT ix - ROW_NUMBER() OVER (ORDER BY ix) as delta, * FROM _
↪transactions) groups
  group by groups.delta
```

(continues on next page)

(continued from previous page)

```
  order by groups.delta
  limit 1;

$$ language sql;
create or replace function first_checkpoint()
returns table ("offset" _transactions."offset"%type, ix _transactions.ix%type) as
↪$$
  select t."offset" as "offset", t."ix" as ix from _transactions t order by ix
↪limit 1;
```

Note that the `Archive` table represents all `Archive` choices in the given namespace. ie. `User.Archive` and `Alias.Archive` in the `User` namespace.

1.46.5.4 JSON Format

PQS stores create and exercise arguments using a Daml-LF JSON-based encoding of Daml-LF values. An overview of the encoding is provided below. For more details, refer to the Daml-LF page.

Values on the ledger can be primitive types, user-defined records, or variants. An extracted contract is represented in the database as a record of its create argument. The fields of that record are primitive types, other records, or variants. A contract can be a recursive structure of arbitrary depth.

These types are translated to JSON types as follows:

Primitive types

`ContractID`: Represented as string.
`Int64`: Represented as string.
`Decimal`: Represented as string.
`List`: Represented as array.
`Text`: Represented as string.
`Date`: Days since the Unix epoch. represented as integer.
`Time`: Microseconds since the UNIX epoch. Represented as number.
`Bool`: Represented as boolean.
`Party`: Represented as string.
`Unit` and `Empty`: Represented as empty records.
`Optional`: Represented as object. It is a Variant with two possible constructors: `None` and `Some`.

User-defined types

`Record`: Represented as object, where each create parameter's name is a key, and the parameter's value is the JSON-encoded value.
`Variant`: Represented as object, using the `{constructor: body}` format, such as `{"Left": true}`.

1.46.5.5 Display of Metadata-Inferred Database Schema

PQS analyzes package metadata as part of its operation and displays the required schema to the user, as shown in the following example

```
$ ./scribe.jar datastore postgres-document schema show
[...]
/************************************************************
* generated by scribe, version: v0.0.1-main+2151-7961ecb *
************************************************************/
-- tables
create table if not exists _transactions (
"offset" text primary key not null,
ix bigint not null,
transaction_id text,
effective_at timestamp with time zone,
workflow_id text
);
[...]
```

or it applies the schema on the fly idempotently (default).

```
$ ./scribe.jar pipeline ledger postgres-document --pipeline-party=Alice
18:27:26.799 I [zio-fiber-64] com.digitalasset.scribe.appversion.package:11
 ↪scribe, version: v0.0.1-main+2151-7961ecb
18:27:27.159 I [zio-fiber-68] com.digitalasset.scribe.configuration.package:40
 ↪Applied configuration:
pipeline {
datasource=TransactionStream
[...]
18:27:28.714 I [zio-fiber-67] com.digitalasset.scribe.postgres.document.
 ↪DocumentPostgres.Service:36 Applying schema
18:27:28.805 I [zio-fiber-67] com.digitalasset.scribe.postgres.document.
 ↪DocumentPostgres.Service:39 Schema applied
18:27:28.863 I [zio-fiber-0] com.digitalasset.scribe.pipeline.pipeline.Impl:29
 ↪Starting pipeline on behalf of
'party-e303d252-1e35-46cb-b4e6-
 ↪06538271d927::1220883670ff44119c947deeabb2e07827adff83bed3e1a897f53f73b0f61d509952
 ↪'
18:27:29.043 I [zio-fiber-0] com.digitalasset.scribe.pipeline.pipeline.Impl:57
 ↪Last checkpoint is absent.
Seeding from ACS before processing transactions with starting offset
 ↪'00000000000000008'
18:27:29.063 I [zio-fiber-938] com.digitalasset.zio.daml.Ledger.Impl:191 Contract
 ↪filter inclusive of 2 templates
and 0 interfaces
18:27:29.120 I [zio-fiber-0] com.digitalasset.scribe.pipeline.pipeline.Impl:74
 ↪Continuing from offset 'GENESIS' and
index '0' until offset 'INFINITY'
18:27:29.159 I [zio-fiber-967] com.digitalasset.zio.daml.Ledger.Impl:191 Contract
 ↪filter inclusive of 2 templates
and 0 interfaces
[...]
```

1.46.5.6 PQS Database Schema

The following schema is representative for the exported ledger data. It is subject to change, since it is an Early Access feature.

```
/*************************************************************
 * generated by scribe, version: v0.0.1-main+2151-7961ecb *
 *************************************************************/
 -- tables
 create table if not exists _transactions (
   "offset" text primary key not null,
   ix bigint not null,
   transaction_id text,
   effective_at timestamp with time zone,
   workflow_id text
 );

 create table if not exists _exercises (
   event_id text primary key not null,
   choice text not null,
   contract_id text not null,
   "offset" text not null references _transactions ("offset") on delete cascade
↪on update cascade,
   consuming bool,
   witnesses text[],
   parent text references _exercises (event_id) on delete cascade
 );

 create table if not exists _creates (
   event_id text primary key not null,
   contract_id text not null,
   "offset" text not null references _transactions ("offset") on delete cascade
↪on update cascade,
   witnesses text[],
   parent text references _exercises (event_id) on delete cascade
 );

 create table if not exists _archives (
   event_id text primary key not null,
   contract_id text not null,
   "offset" text not null references _transactions ("offset") on delete cascade
↪on update cascade
 );

 create table if not exists _mappings (
   daml_fqn text primary key not null,
   pg_identifier text not null unique
 );

 -- PAYLOAD TABLES
 create table if not exists "Alias.39p75i" (
   event_id text primary key not null references _creates (event_id) on delete
↪cascade,
   identifier text not null,
   contract_key jsonb,
   payload jsonb not null
 );
```

(continues on next page)

(continued from previous page)

```
create table if not exists "User.11jk59n1" (
  event_id text primary key not null references _creates (event_id) on delete
↪cascade,
  identifier text not null,
  contract_key jsonb,
  payload jsonb not null
);

create table if not exists "Archive.2gpwea" (
  event_id text primary key not null references _exercises (event_id) ondelete
↪cascade,
  identifier text not null,
  argument jsonb not null,
  result jsonb not null
);

create table if not exists "Alias_Change.11wa21n1" (
  event_id text primary key not null references _exercises (event_id) on delete
↪cascade,
  identifier text not null,
  argument jsonb not null,
  result jsonb not null
);

create table if not exists "User_Follow.11q646ez" (
  event_id text primary key not null references _exercises (event_id) on delete
↪cascade,
  identifier text not null,
  argument jsonb not null,
  result jsonb not null
);
```

Note that the Archive table represents all Archive choices in the given namespace, such as `User.Archive` and `Alias.Archive` in the User namespace.

1.46.6 Operating PQS

This section discusses the common tasks to perform when operating a PQS.

1.46.6.1 Purging Excessive Historical Ledger Data

Pruning ledger data from the PQS database can help reduce storage size and improve query performance by removing old data. PQS provides two approaches to prune ledger data: using the PQS CLI or using the `prune_to_offset` PostgreSQL function.

WARNING: Calling either the `prune` CLI command with `--prune-mode Force`, or calling the PostgreSQL function `prune_to_offset` will delete data irrevocably.

Both pruning approaches (CLI and PostgreSQL function) share the same behavior in terms of data deletion and changes:

Active contracts are preserved under a new offset, while all other transaction-related data up to, and including the target offset is deleted.

The target offset, ie. the offset provided via `--prune-target` or as argument to `prune_to_offset`, is the transaction with the highest offset that will be deleted by the pruning operation.

Note: If the provided offset (i.e. via `--prune-target`, or as argument to `prune_to_offset`) does not have a transaction record, then the effective target offset will be the oldest transaction offset that succeeds (is greater than) the provided offset.

When using either pruning method, the following data will be changed:

> The offset of active contracts will be moved to the oldest known offset which succeeds the pruning target offset, i.e. the offset of the oldest transaction that is unaffected by the pruning operation.

The following data will be deleted:

> Transactions with offsets up to and including the target offset.
> Events, archived contracts and exercise payloads associated with the deleted transactions.

The following data will be unaffected:

> Transaction related data (event, choices, or contracts) for transaction with an offset that is greater than the effective pruning target offset.

Pruning is a destructive operation and cannot be undone. If necessary, make sure to backup your data before performing any pruning operations.

There are some constraints when using either pruning method:

1. The provided target offset must be within the bounds of the contiguous history. If the target offset is outside the bounds, an error will be raised.
2. The pruning operation cannot coincide with the latest consistent checkpoint of the contiguous history. If it does, an error will be raised.

Pruning with PQS CLI

The PQS CLI provides a `prune` command that allows you to prune the ledger data up to a specified offset, timestamp, or duration.

For detailed information on all available options, please run `./PQS.jar datastore postgres-document prune --help-verbose`.

To use the `prune` command, you need to provide a pruning target as an argument. The pruning target can be an offset, a timestamp (ISO 8601), or a duration (ISO 8601):

```
./PQS.jar datastore postgres-document prune --prune-target <pruning_target>
```

By default, the `prune` command performs a dry run, which means it will only display the effects of the pruning operation without actually deleting any data. To execute the pruning operation, you need to add the `--prune-mode Force` option:

```
./PQS.jar datastore postgres-document prune --prune-target <pruning_target> --
↪prune-mode Force
```

Instead of providing an offset as the `--prune-target`, you can use a timestamp or duration as the pruning cutoff. For example, the following command prunes data older than 30 days (relative to now):

```
./PQS.jar datastore postgres-document prune --prune-target P30D
```

The following example prunes data up to a specific timestamp:

```
./PQS.jar datastore postgres-document prune --prune-target 2023-01-30T00:00:00.
↪000Z
```

Pruning with `prune_to_offset`

The `prune_to_offset` PostgreSQL function allows you to prune ledger data up to a specified offset. It has the same behavior as the `datastore postgres-document prune` command, except it does not offer dry runs.

To use `prune_to_offset`, provide an offset as a text argument:

```
SELECT * FROM prune_to_offset('<offset>');
```

This function deletes transactions and update active contracts as described *earlier in this section*.

To prune data up to a specific timestamp or interval, use `prune_to_offset` in combination with the `get_offset` function. For example, the following query prunes data older than 30 days:

```
SELECT * FROM prune_to_offset(get_offset(interval '30 days'));
```

1.46.7 Optimizing PQS

This section briefly discusses optimizing a database as an introduction. The topic is broad, and there are many resources available. Refer to the PostgreSQL documentation for more information.

1.46.7.1 Indexing

indexes are an important tool to make queries with (JSON) expressions perform well. Here is one example of an index:

```
CREATE INDEX issueDateIdx
ON contract
USING BTREE ((payload->'issuanceData'->'issueDate'->>'Some'));
```

In this example, the index allows comparisons on the issue date. It has the additional advantage that the results of the JSON query `payload->'issuanceData'->'issueDate'->>'Some'` are cached and do not have to be recomputed for every access.

PostgreSQL provides several index types, including B-tree, Hash, GiST, SP-GiST, GIN, and BRIN. Each index type uses a different algorithm that is best suited to different types of queries. The table below provides a basic explanation of where they can be used. For a more thorough understanding, consult the chapter on indexes in the PostgreSQL manual.

Index Type	Comment
Hash	Compact. Useful only for filters that use =.
B-tree	Can be used in filters that use <, <=, =, >=, > as well as postfix string comparisons (e.g. LIKE 'foo%'). B-trees can also speed up ORDER BY clauses and can be used to retrieve subexpressions values from the index rather than evaluating the subexpressions (i.e. when used in a SELECT clause).
GIN	Useful for subset operators.
GiST, SP-GiST	See manual.
BRIN	Efficient for tables where rows are already physically sorted for a particular column.

1.46.7.2 Pagination

Pagination refers to splitting up large result sets into pages of up to `n` results. It can allow user navigation such as moving to the next page to display, going to the end of the result set, or jumping around in the middle. It can be a very effective user experience when there is a large ordered data set. The following pagination use cases are important:

Pagination Use Case		Example
Random access	Accessing arbitrary pages	**Client side binary search of** results. **A user opens random pages in a** search result.
Iteration or enumeration	Accessing page 1, then page 2,	**Programmatic processing of all** results in batches.

For efficient pagination iteration, you first need a column to sort on. The requirements are:

1. It should be acceptable to the user to sort results on this column.
2. You need a (unique) B-tree index on this column.
3. The column must have unique values.

You can then perform queries like this:

```
SELECT *
FROM the_table
WHERE the_sort_col > ???
ORDER BY the_sort_col
LIMIT 100;
```

The `???` value represents the last (largest) value for `the_sort_col` that was previously returned. To fetch results for the very first page, omit the `WHERE` clause.

Here is an example of random access to display page 10 of the search results:

```
SELECT *
FROM the_table
ORDER BY the_sort_col
```

(continues on next page)

(continued from previous page)

```
LIMIT 100
OFFSET 1000;
```

This only makes sense if there is a B-tree index on `the_sort_col`.

You should assume that a large `OFFSET` is slow. See the chapter on LIMIT and OFFSET in the PostgreSQL manual.

1.46.7.3 psql Tips

Type `psql <dbname>` on the command line to enter the PostgreSQL `REPL` (if in doubt, use postgres as the database name). Some useful commands are shown in the following table.

Command	Description
\l	List all databases.
\c db	Switch to a different database.
\d	List all tables in the current database.
\d table	Show a table, including column types and indexes.

To create databases and users, try this:

```
CREATE DATABASE the_db;
CREATE USER the_user WITH PASSWORD 'abc123';
```

To later remove them, try this:

```
DROP DATABASE the_db;
DROP USER the_user;
```

psql can also be used for scripting:

```
psql postgres <<END
...
CREATE DATABASE the_db;
...
END
```

The script continues to execute if a command fails.

1.46.7.4 EXPLAIN ANALYZE

Type `EXPLAIN ANALYZE` followed by a query in `psql` or similar tools to get an explanation of how the query would be executed. This is an invaluable tool to verify that a query you might want to run uses the indexes that you think it does.

```
EXPLAIN ANALYZE
SELECT COUNT(*) FROM the_table;
```

1.46.8 Troubleshooting

Some of the most common troubleshooting options are discussed below.

1.46.8.1 Cannot Connect to the Ledger Node

If the PQS cannot connect to the ledger node on startup, you will see a message in the logs like the following example, and the PQS will terminate.

```
21:15:02.084 E [zio-fiber-0] com.digitalasset.scribe.app.ComposableApp:34
↪Exception in thread
"zio-fiber-" io.grpc.StatusException: UNAVAILABLE: io exception
  at
scalapb.zio_grpc.client.UnaryClientCallListener.onClose$$anonfun$1$$anonfun
↪$1(UnaryClientCallListener.scala:61)
  Suppressed:
io.netty.channel.AbstractChannel$AnnotatedConnectException: Connection refused:
localhost/[0:0:0:0:0:0:0:1]:6865
    Suppressed: java.net.ConnectException: Connection refused
      at java.base/sun.nio.ch.Net.pollConnect(Native Method)
      at java.base/sun.nio.ch.Net.pollConnectNow(Net.java:672)
      at java.base/sun.nio.ch.SocketChannelImpl.finishConnect(SocketChannelImpl.
↪java:946)
      at io.netty.channel.socket.nio.NioSocketChannel.
↪doFinishConnect(NioSocketChannel.java:337)
      at io.netty.channel.nio.AbstractNioChannel$AbstractNioUnsafe.
↪finishConnect(AbstractNioChannel.java:334)
      at io.netty.channel.nio.NioEventLoop.processSelectedKey(NioEventLoop.
↪java:776)
      at io.netty.channel.nio.NioEventLoop.
↪processSelectedKeysOptimized(NioEventLoop.java:724)
      at io.netty.channel.nio.NioEventLoop.processSelectedKeys(NioEventLoop.
↪java:650)
      at io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:562)
      at io.netty.util.concurrent.SingleThreadEventExecutor$4.
↪run(SingleThreadEventExecutor.java:997)
      at io.netty.util.internal.ThreadExecutorMap$2.run(ThreadExecutorMap.java:74)
      at io.netty.util.concurrent.FastThreadLocalRunnable.
↪run(FastThreadLocalRunnable.java:30)
      at java.base/java.lang.Thread.run(Thread.java:833)
io.grpc.StatusException: UNAVAILABLE: io exception
io.netty.channel.AbstractChannel.AnnotatedConnectException: Connection
refused: localhost/[0:0:0:0:0:0:0:1]:6865
java.net.ConnectException: Connection refused
```

To fix this, make sure that the participant node's ledger API is accessible from where you are running the PQS.

1.46.8.2 Cannot Connect to the PQS Database

If the database is not available before the transaction stream is started, the PQS will terminate and you will see as error from the JDBC driver in the logs similar to the following example.

```
21:16:32.116 E [zio-fiber-0] com.digitalasset.scribe.app.ComposableApp:34
↪Exception in thread
"zio-fiber-" org.postgresql.util.PSQLException: Connection to localhost:5432
↪refused. Check
that the hostname and port are correct and that the postmaster is accepting TCP/
↪IP connections.
  at
    org.postgresql.core.v3.ConnectionFactoryImpl.
↪openConnectionImpl(ConnectionFactoryImpl.java:342)
  at org.postgresql.core.ConnectionFactory.openConnection(ConnectionFactory.
↪java:54)
  at org.postgresql.jdbc.PgConnection.<init>(PgConnection.java:263)
  at org.postgresql.Driver.makeConnection(Driver.java:443)
  at org.postgresql.Driver.connect(Driver.java:297)
  at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:681)
  at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:190)
  at zio.jdbc.shims.postgres$.$anonfun$1(postgres.scala:21)
  at
    zio.ZIOCompanionVersionSpecific.attempt$$anonfun
↪$1(ZIOCompanionVersionSpecific.scala:103)
  at zio.ZIO$.suspendSucceed$$anonfun$1(ZIO.scala:4589)
  at
    zio.UnsafeVersionSpecific.implicitFunctionIsFunction$$anonfun
↪$1(UnsafeVersionSpecific.scala:27)
  at zio.Unsafe$.unsafe(Unsafe.scala:37)
  at zio.ZIOCompanionVersionSpecific.succeed$$anonfun
↪$1(ZIOCompanionVersionSpecific.scala:185)
  Suppressed: java.net.ConnectException: Connection refused
    at java.base/sun.nio.ch.Net.pollConnect(Native Method)
    at java.base/sun.nio.ch.Net.pollConnectNow(Net.java:672)
    at java.base/sun.nio.ch.NioSocketImpl.timedFinishConnect(NioSocketImpl.
↪java:547)
    at java.base/sun.nio.ch.NioSocketImpl.connect(NioSocketImpl.java:602)
    at java.base/java.net.SocksSocketImpl.connect(SocksSocketImpl.java:327)
    at java.base/java.net.Socket.connect(Socket.java:633)
    at org.postgresql.core.PGStream.createSocket(PGStream.java:243)
    at org.postgresql.core.PGStream.<init>(PGStream.java:98)
    at org.postgresql.core.v3.ConnectionFactoryImpl.
↪tryConnect(ConnectionFactoryImpl.java:132)
    at
      org.postgresql.core.v3.ConnectionFactoryImpl.
↪openConnectionImpl(ConnectionFactoryImpl.java:258)
    at org.postgresql.core.ConnectionFactory.openConnection(ConnectionFactory.
↪java:54)
    at org.postgresql.jdbc.PgConnection.<init>(PgConnection.java:263)
    at org.postgresql.Driver.makeConnection(Driver.java:443)
    at org.postgresql.Driver.connect(Driver.java:297)
    at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:681)
    at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:190)
    at zio.jdbc.shims.postgres$.$anonfun$1(postgres.scala:21)
    at
      zio.ZIOCompanionVersionSpecific.attempt$$anonfun
↪$1(ZIOCompanionVersionSpecific.scala:103)
```

(continues on next page)

(continued from previous page)

```
    at zio.ZIO$.suspendSucceed$$anonfun$1(ZIO.scala:4589)
    at
      zio.UnsafeVersionSpecific.implicitFunctionIsFunction$$anonfun
↪$1(UnsafeVersionSpecific.scala:27)
    at zio.Unsafe$.unsafe(Unsafe.scala:37)
    at
      zio.ZIOCompanionVersionSpecific.succeed$$anonfun
↪$1(ZIOCompanionVersionSpecific.scala:185)
org.postgresql.util.PSQLException: Connection to localhost:5432 refused. Check
↪that
the hostname and port are correct and that the postmaster is accepting TCP/IP
↪connections.
java.net.ConnectException: Connection refused
```

To fix this, make sure that the database exists and is accessible from where you are running the PQS. Also, ensure that the database username and password are correct and that the credentials to connect to the database from the network address are set properly.

If the database connection is broken while the transaction stream was already running, you will see a similar message in the logs, but it will be repeated. The transaction stream will be restarted with an exponential backoff. This gives the database, network, or any other troubled resource time to get back into shape. Once everything is in order, the stream will continue without any need for manual intervention.

1.47 Daml Ledger Interoperability

Certain Daml ledgers can interoperate with other Daml ledgers. That is, the contracts created on one ledger can be used and archived in transactions on other ledgers. Some Participant Nodes can connect to multiple ledgers and provide their parties unified access to those ledgers via the *Ledger API*. For example, when an organization initially deploys two workflows to two Daml ledgers, it can later compose those workflows into a larger workflow that spans both ledgers.

Interoperability may limit the visibility a Participant Node has into a party's ledger projection, i.e., its *local ledger*, when the party is hosted on multiple Participant Nodes. These limitations influence what parties can observe via the Ledger API of each Participant Node. In particular, interoperability affects which events a party observes and their order. This document explains the visibility limitations due to interoperability and their consequences for the Transaction Service, by *example* and formally by introducing interoperable versions of *causality graphs* and *projections*.

The presentation assumes that you are familiar with the following concepts:

The *Ledger API*
The *Daml Ledger Model*
Local ledgers and causality graphs

Note: Interoperability for Daml ledgers is under active development. This document describes the vision for interoperability and gives an idea of how the Ledger API services may change and what guarantees are provided. The described services and guarantees may change without notice as the interoperability implementation proceeds.

1.47.1 Interoperability Examples

1.47.1.1 Topology

Participant Nodes connect to Daml ledgers and parties access projections of these ledgers via the Ledger API. The following picture shows such a setup.

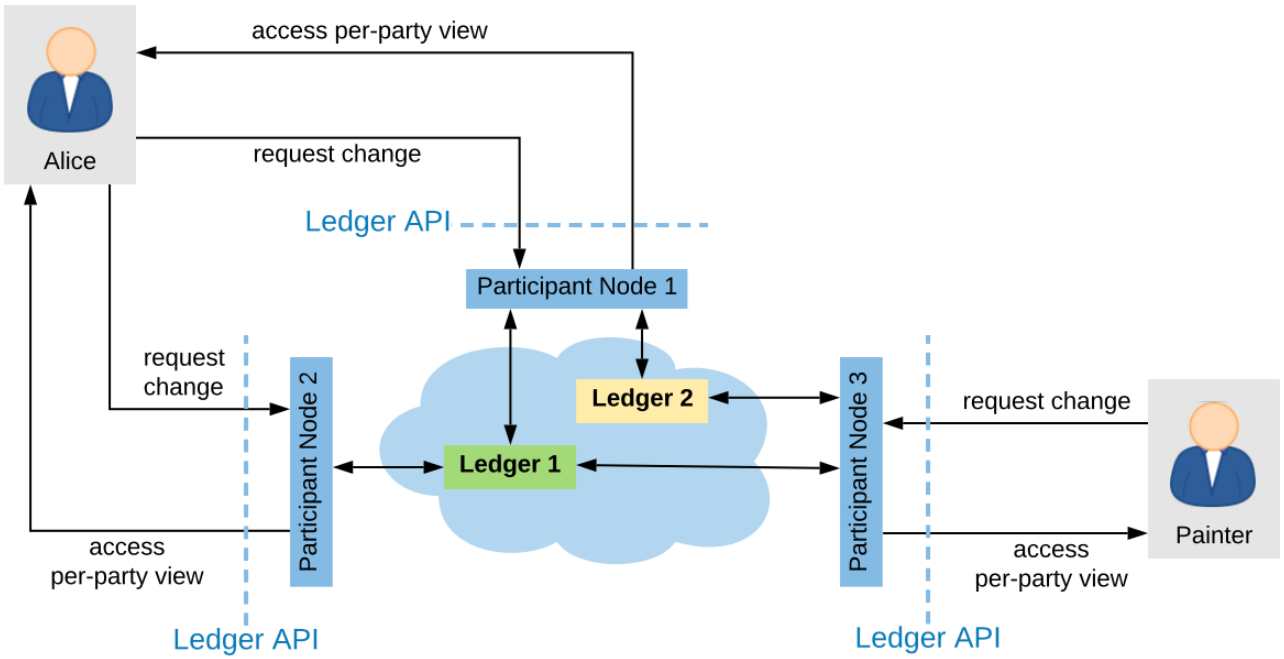


Fig. 34: Example topology with two interoperable ledgers

The components in this diagram are the following:

There is a set of interoperable **Daml ledgers**: Ledger 1 (green) and Ledger 2 (yellow).
Each **Participant Node** is connected to a subset of the Daml ledgers.

- Participant Nodes 1 and 3 are connected to Ledger 1 and 2.
- Participant Node 2 is connected to Ledger 1 only.

Participant Nodes host parties on a subset of the Daml ledgers they are connected to. A Participant Node provides a party access to the Daml ledgers that it hosts the party on.

- Participant Node 1 hosts Alice on Ledger 1 and 2.
- Participant Node 2 hosts Alice on Ledger 1.
- Participant Node 3 hosts the painter on Ledger 1 and 2.

1.47.1.2 Aggregation at the Participant

The Participant Node assembles the updates from these ledgers and outputs them via the party's Transaction Service and Active Contract Service. When a Participant Node hosts a party only on a subset of the interoperable Daml ledgers, then the transaction and active contract services of the Participant Node are derived only from those ledgers.

For example, in the *above topology*, when a transaction creates a contract with stakeholder Alice on Ledger 2, then *P1*'s transaction stream for Alice will emit this transaction and report the contract as active, but Alice's stream at *P2* will not.

1.47.1.3 Enter and Leave Events

With interoperability, a transaction can use a contract whose creation was recorded on a different ledger. In the *above topology*, e.g., one transaction creates a contract *c1* with stakeholder Alice on Ledger 1 and another archives the contract on Ledger 2. Then the Participant Node *P2* outputs the **Create** action as a `CreatedEvent`, but not the **Exercise** in form of an `ArchiveEvent` on the transaction service because Ledger 2 can not notify *P2* as *P2* does not host Alice on Ledger 2. Conversely, when one transaction creates a contract *c2* with stakeholder Alice on Ledger 2 and another archives the contract on Ledger 1, then *P2* outputs the `ArchivedEvent`, but not the `CreatedEvent`.

To keep the transaction stream consistent, *P2* additionally outputs a **Leave** *c1* action on Alice's transaction stream. This action signals that the Participant Node no longer outputs events concerning this contract; in particular not when the contract is archived. The contract is accordingly no longer reported in the active contract service and cannot be used by command submissions.

Conversely, *P2* outputs an **Enter** *c2* action some time before the `ArchivedEvent` on the transaction stream. This action signals that the Participant Node starts outputting events concerning this contract. The contract is reported in the Active Contract Service and can be used by command submission.

The actions **Enter** and **Leave** are similar to a **Create** and a consuming **Exercise** action, respectively, except that **Enter** and **Leave** may occur several times for the same contract whereas there should be at most one **Create** action and at most one consuming **Exercise** action for each contract.

These **Enter** and **Leave** events are generated by the underlying interoperability protocol. This may happen as part of command submission or for other reasons, e.g., load balancing. It is guaranteed that the **Enter** action precedes contract usage, subject to the trust assumptions of the underlying ledgers and the interoperability protocol.

A contract may enter and leave the visibility of a Participant Node several times. For example, suppose that the painter submits the following commands and their commits end up on the given ledgers.

1. Create a contract *c* with signatories Alice and the painter on Ledger 2
2. Exercise a non-consuming choice *ch1* on *c* on Ledger 1.
3. Exercise a non-consuming choice *ch2* on *c* on Ledger 2.
4. Exercise a consuming choice *ch3* on *c* on Ledger 1.

Then, the transaction tree stream that *P2* provides for *A* contains five actions involving contract *c*: **Enter**, non-consuming **Exercise**, **Leave**, **Enter**, consuming **Exercise**. Importantly, *P2* must not omit the **Leave** action and the subsequent **Enter**, even though they seem to cancel out. This is because their presence indicates that *P2*'s event stream for Alice may miss some events in between; in this example, exercising the choice *ch2*.

The flat transaction stream by *P2* omits the non-consuming exercise choices. It nevertheless contains the three actions **Enter**, **Leave**, **Enter** before the consuming **Exercise**. This is because the Participant Node cannot know at the **Leave** action that there will be another **Enter** action coming.

In contrast, *P1* need not output the **Enter** and **Leave** actions at all in this example because *P1* hosts Alice on both ledgers.

1.47.1.4 Cross-ledger Transactions

With interoperability, a cross-ledger transaction can be committed on several interoperable Daml ledgers simultaneously. Such a cross-ledger transaction avoids some of the synchronization overhead of **Enter** and **Leave** actions. When a cross-ledger transaction uses contracts from several Daml ledgers, stakeholders may witness actions on their contracts that are actually not visible on the Participant Node.

For example, suppose that the *split paint counteroffer workflow* from the causality examples is committed as follows: The actions on *CounterOffer* and *PaintAgree* contracts are committed on Ledger 1. All actions on *Iou*s are committed on Ledger 2, assuming that some Participant Node hosts the Bank on Ledger 2. The last transaction is a cross-ledger transaction because the archival of the *CounterOffer* and the creation of the *PaintAgree*ment commits on Ledger 1 simultaneously with the transfer of Alice's *Iou* to the painter on Ledger 2.

For the last transaction, Participant Node 1 notifies Alice of the transaction tree, the two archivals and the *PaintAgree* creation via the Transaction Service as usual. Participant Node 2 also output's the whole transaction tree on Alice's transaction tree stream, which contains the consuming **Exercise** of Alice's *Iou*. However, it has not output the **Create** of Alice's *Iou* because *Iou* actions commit on Ledger 2, on which Participant Node 2 does not host Alice. So Alice merely *witnesses* the archival even though she is an *informee* of the exercise. The **Exercise** action is therefore marked as merely being witnessed on Participant Node 2's transaction tree stream.

In general, an action is marked as **merely being witnessed** when a party is an informee of the action, but the action is not committed on a ledger on which the Participant Node hosts the party. Unlike **Enter** and **Leave**, such witnessed actions do not affect causality from the participant's point of view and therefore provide weaker ordering guarantees. Such witnessed actions show up neither in the flat transaction stream nor in the Active Contracts Service.

For example, suppose that the **Create** *PaintAgree* action commits on Ledger 2 instead of Ledger 1, i.e., only the *CounterOffer* actions commit on Ledger 1. Then, Participant Node 2 marks the **Create** *PaintAgree* action also as merely being witnessed on the transaction tree stream. Accordingly, it does not report the contract as active nor can Alice use the contract in her submissions via Participant Node 2.

1.47.2 Multi-ledger Causality Graphs

This section generalizes *causality graphs* to the interoperability setting.

Every active Daml contract resides on at most one Daml ledger. Any use of a contract must be committed on the Daml ledger where it resides. Initially, when the contract is created, it takes up residence on the Daml ledger on which the **Create** action is committed. To use contracts residing on different Daml ledgers, cross-ledger transactions are committed on several Daml ledgers.

However, cross-ledger transactions incur overheads and if a contract is frequently used on a Daml ledger that is not its residence, the interoperability protocol can migrate the contract to the other

Daml ledger. The process of the contract giving up residence on the origin Daml ledger and taking up residence on the target Daml ledger is called a **contract transfer**. The **Enter** and **Leave** events on the transaction stream originate from such contract transfers, as will be explained below. Moreover, contract transfers are synchronization points between the origin and target Daml ledgers and therefore affect the ordering guarantees. We therefore generalize causality graphs for interoperability.

Definition Transfer action A **transfer action** on a contract *c* is written **Transfer** *c*. The **informees** of the transfer actions are the stakeholders of *c*.

In the following, the term *action* refers to transaction actions (**Create**, **Exercise**, **Fetch**, and **NoSuchKey**) as well as transfer actions. In particular, a transfer action on a contract *c* is an action on *c*. Transfer actions do not appear in transactions though. So a transaction action cannot have a transfer action as a consequence and transfer actions do not have consequences at all.

Definition Multi-Ledger causality graph A **multi-ledger causality graph** *G* for a set *Y* of Daml ledgers is a finite, transitively closed, directed acyclic graph. The vertices are either transactions or transfer actions. Every action is possibly annotated with an **incoming ledger** and an **outgoing ledger** from *Y* according to the following table:

Action	incoming ledger	outgoing ledger
Create	no	yes
consuming **Exercise**	yes	no
non-consuming **Exercise**	yes	yes
Fetch	yes	yes
NoSuchKey	no	no
Transfer	maybe	maybe

For non-consuming **Exercise** and **Fetch** actions, the incoming ledger must be the same as the outgoing ledger. **Transfer** actions must have at least one of them. A **transfer** action with both set represents a complete transfer. If only the incoming ledger is set, it represents the partial information of an **Enter** event; if only outgoing is set, it is the partial information of a **Leave** event. **Transfer** actions with missing incoming or outgoing ledger annotations referred to as **Enter** or **Leave** actions, respectively.

The *action order* generalizes to multi-ledger causality graphs accordingly.

In the *example for Enter and Leave events* where the painter exercises three choices on contract *c* with signatories Alice and the painter, the four transactions yield the following multi-ledger causality graph. Incoming and outgoing ledgers are encoded as colors (green for Ledger 1 and yellow for Ledger 2). **Transfer** vertices are shown as circles, where the left half is colored with the incoming ledger and the right half with the outgoing ledger.

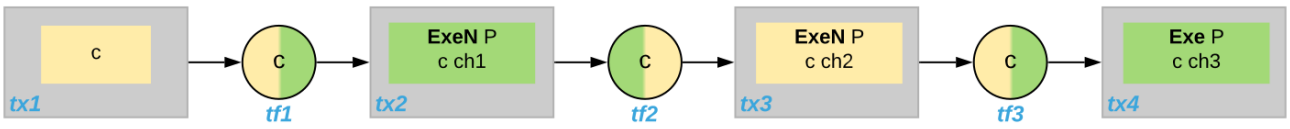


Fig. 35: Multi-Ledger causality graph with transfer actions

Note: As for ordinary causality graphs, the diagrams for multi-ledger causality graphs omit transitive edges for readability.

As an example for a cross-domain transaction, consider the *split paint counteroffer workflow with the cross-domain transaction*. The corresponding multi-ledger causality graph is shown below. The last transaction *tx4* is a cross-ledger transaction because its actions have more than one color.

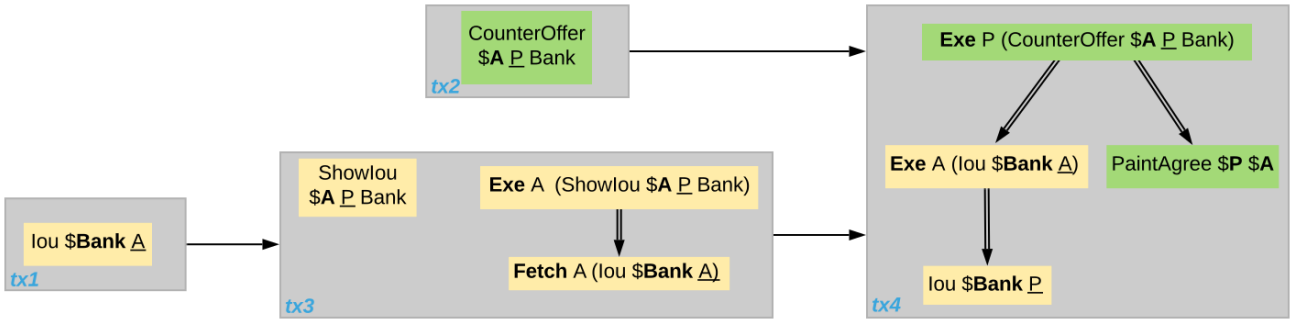


Fig. 36: Multi-Ledger causality graph for the split paint counteroffer workflow on two Daml ledgers

1.47.2.1 Consistency

Definition **Ledger trace** A **ledger trace** is a finite list of pairs (a_i, b_i) such that $b_{i-1} = a_i$ for all $i > 0$. Here a_i and b_i identify Daml ledgers or are the special value *NONE*, which is different from all Daml ledger identifiers.

Definition **Multi-Ledger causal consistency for a contract** Let *G* be a multi-ledger causality graph and *X* be a set of actions from *G* on a contract in *c*. The graph *G* is **multi-ledger consistent for the contract** *c* on *X* if all of the following hold:

1. If *X* is not empty, then *X* contains a **Create** or at least one **Enter** action. If it contains a create, then this create precedes all other actions in *X*. If it does not, then there exists one **Enter** action that precedes all other actions in *X*.
2. *X* contains at most one **Create** action.
3. If *X* contains a consuming **Exercise** action *act*, then *act* follows all other actions in *X* in *G*'s action order.
4. All **Transfer** actions in *X* are ordered with all other actions in *X*.
5. For every maximal chain in *X* (i.e., maximal totally ordered subset of *X*), the sequence of (incoming ledger, outgoing ledger) pairs is a ledger trace, using *NONE* if the action does not have an incoming or outgoing ledger annotation.

The first three conditions mimic the conditions of *causal consistency* for ordinary causality graphs. They ensure that **Create** actions come first and consuming **Exercise** actions last. An **Enter** action takes the role of a **Create** if there is no **Create**. The fourth condition ensures that all transfer actions are synchronization points for a contract. The last condition about ledger traces ensures that contracts reside on only one Daml ledger and all usages happen on the ledger of residence. In particular, the next contract action after a **Leave** must be an **Enter**.

For example, the above *multi-ledger causality graph with transfer actions* is multi-ledger consistent for *c*. In particular, there is only one maximal chain in the actions on *c*, namely

> **Create** *c* -> *tf1* -> **ExeN** *B c ch1* -> *tf2* -> **ExeN** *B c ch2* -> *tf3* -> **ExeN** *B c ch3*,

and for each edge act_1 -> act_2, the outgoing ledger color of act_1 is the same as the incoming ledger color of act_2. The restriction to maximal chains ensures that no node is skipped. For example, the (non-maximal) chain

Create *c* -> **ExeN** *B c ch1* -> *tf2* -> **ExeN** *B c ch2* -> *tf3* -> **Exe** *B c ch3*

is not a ledger trace because the outgoing ledger of the **Create** action (yellow) is not the same as the incoming ledger of the non-consuming **Exercise** action for *ch1* (green). Accordingly, the subgraph without the *tf1* vertex is not multi-ledger consistent for *c* even though it is a multi-ledger causality graph.

Definition **Consistency for a multi-ledger causality graph** Let *X* be a subset of actions in a multi-ledger causality graph *G*. Then *G* is **multi-ledger consistent** for *X* (or *X*-**multi-ledger consistent**) if *G* is multi-ledger consistent for all contracts *c* on the set of actions on *c* in *X*. *G* is **multi-ledger consistent** if *G* is multi-ledger consistent on all the actions in *G*.

Note: There is no multi-ledger consistency requirement for contract keys yet. So interoperability does not provide consistency guarantees beyond those that come from the contracts they reference. In particular, contract keys need not be unique and **NoSuchKey** actions do not check that the contract key is unassigned.

The *multi-ledger causality graph for the split paint counteroffer workflow* is multi-ledger consistent. In particular all maximal chains of actions on a contract are ledger traces:

contract	maximal chains
Iou Bank A	**Create** -> **Fetch** -> **Exercise**
ShowIou A P Bank	**Create** -> **Exercise**
Counteroffer A P Bank	**Create** -> **Exercise**
Iou Bank P	**Create**
PaintAgree P A	**Create**

1.47.2.2 Minimality and Reduction

When edges are added to an *X*-multi-ledger consistent causality graph such that it remains acyclic and transitively closed, the resulting graph is again *X*-multi-ledger consistent. The notions *minimally consistent* and *reduction* therefore generalize from ordinary causality graphs accordingly.

Definition **Minimal multi-ledger-consistent causality graph** An *X*-multi-ledger consistent causality graph *G* is *X*-**minimal** if no strict subgraph of *G* (same vertices, fewer edges) is an *X*-multi-ledger consistent causality graph. If *X* is the set of all actions in *G*, then *X* is omitted.

Definition **Reduction of a multi-ledger consistent causality graph** For an *X*-multi-ledger consistent causality graph *G*, there exists a unique minimal *X*-multi-ledger consistent causality graph $reduce_X(G)$ with the same vertices and the edges being a subset of *G*. $reduce_X(G)$ is called the *X*-**reduction** of *G*. As before, *X* is omitted if it contains all actions in *G*.

Since multi-ledger causality graphs are acyclic, their vertices can be sorted topologically and the resulting list is again a causality graph, where every vertex has an outgoing edge to all later vertices. If the original causality graph is *X*-consistent, then so is the topological sort, as topological sorting merely adds edges.

1.47.2.3 From Multi-ledger Causality Graphs to Ledgers

Multi-Ledger causality graphs *G* are linked to ledgers *L* in the Daml Ledger Model via topological sort and reduction.

> Given a multi-ledger causality graph *G*, drop the incoming and outgoing ledger annotations and all transfer vertices, topologically sort the transaction vertices, and extend the resulting list of transactions with the requesters to obtain a sequence of commits *L*.
> Given a sequence of commits *L*, use the transactions as vertices and add an edge from *tx1* to *tx2* whenever *tx1*'s commit precedes *tx2*'s commit in the sequence. Then add transfer vertices and incoming and outgoing ledger annotations as needed and connect them with edges to the transaction vertices.

This link preserves consistency only to some extent. Namely, if a multi-ledger causality graph is multi-ledger consistent for a contract *c*, then the corresponding ledger is consistent for the contract *c*, too. However, a multi-ledger-consistent causality graph does not yield a consistent ledger because key consistency may be violated. Conversely, a consistent ledger does not talk about the incoming and outgoing ledger annotations and therefore cannot enforce that the annotations are consistent.

1.47.3 Ledger-aware Projection

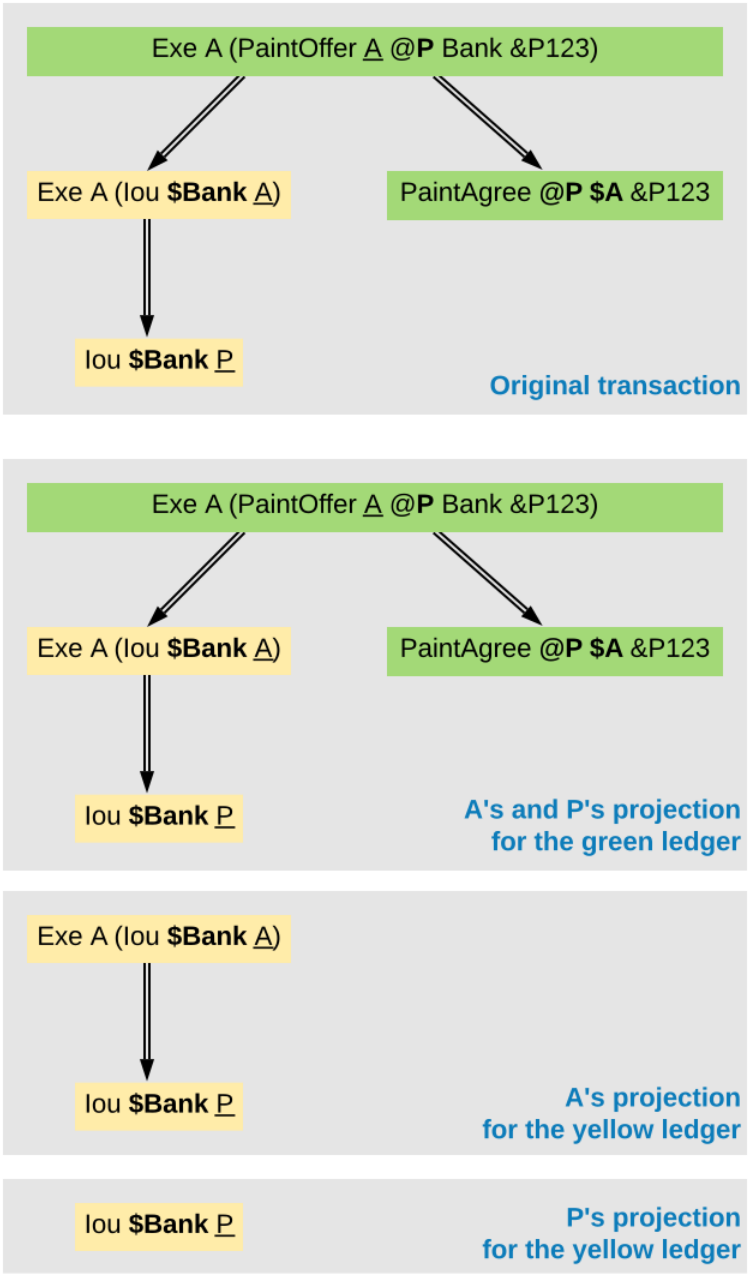
A Participant Node maintains a local ledger for each party it hosts and the Transaction Service outputs a topological sort of this local ledger. When the Participant Node hosts the party on several ledgers, this local ledger is an multi-ledger causality graph. This section defines the ledger-aware projection of an multi-ledger causality graph, which yields such a local ledger.

Definition Y-labelled action An action with incoming and outgoing ledger annotations is **Y-labelled** for a set *Y* if its incoming or outgoing ledger annotation is an element of *Y*.

Definition Ledger-aware projection for transactions Let *Y* be a set of Daml ledgers and *tx* a transaction whose actions are annotated with incoming and outgoing ledgers. Let *Act* be the set of *Y*-labelled subactions of *tx* that the party *P* is an informee of. The **ledger-aware projection** of *tx* for *P* on *Y* (*P*-**projection on** *Y*) consists of all the maximal elements of *Act* (w.r.t. the subaction relation) in execution order.

Note: Every action contains all its subactions. So if *act* is included in the *P*-projection on *Y* of *tx*, then all subactions of *act* are also part of the projection. Such a subaction *act'* may not be *Y*-labelled itself though, i.e., belong to a different ledger. If *P* is an informee of *act'*, the Participant Node will mark *act'* as merely being witnessed on *P*'s transaction stream, as explained below.

The *cross-domain transaction in the split paint counteroffer workflow*, for example, has the following projections for Alice and the painter on the *Iou* ledger (yellow) and the painting ledger (green). Here, the projections on the green ledger include the actions of the yellow ledger because a projection includes the subactions.



Definition **Projection for transfer actions** Let *act* be a transfer action annotated with an incoming ledger and/or an outgoing ledger. The **projection** of *act* on a set of ledgers *Y* removes the annotations from *act* that are not in *Y*. If the projection removes all annotations, it is empty. The **projection** of *act* to a party *P* on *Y* (*P*-**projection** on *Y*) is the projection of *act* on *Y* if *P* is a stakeholder of the contract, and empty otherwise.

Definition **Multi-Ledger consistency for a party** An multi-ledger causality graph *G* is **consistent for a party** *P* on a set of ledgers *Y* (*P*-**consistent** on *Y*) if *G* is multi-ledger consistent on the set of *Y*-labelled actions in *G* of which *P* is a stakeholder informee.

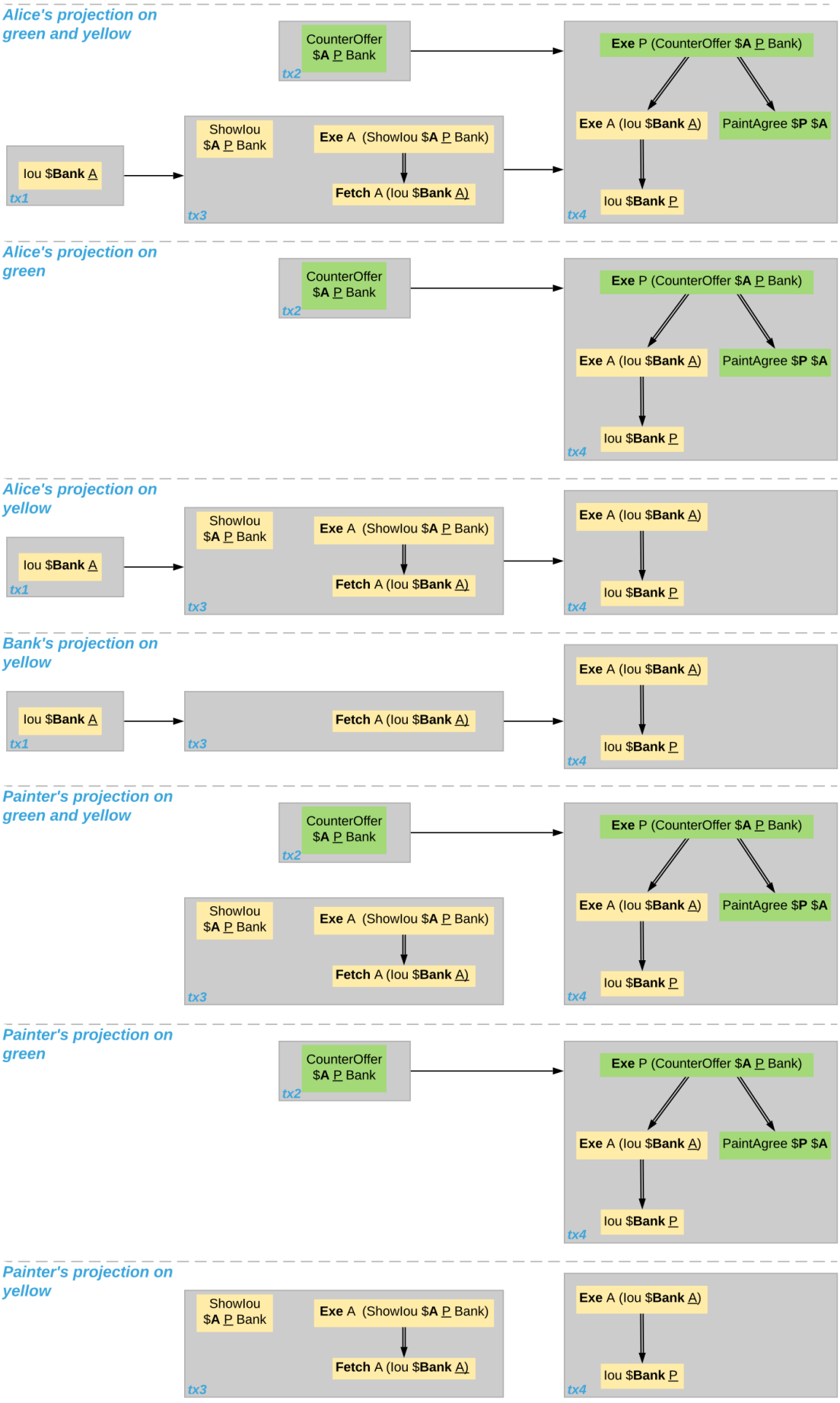
The notions of *X*-minimality and *X*-reduction extend to a party *P* on a set *Y* of ledgers accordingly.

Definition **Ledger-aware projection for multi-ledger causality graphs** Let *G* be a multi-ledger consistent causality graph and *Y* be a set of Daml ledgers. The **projection** of *G* to party *P* on *Y* (*P*-**projection** on *Y*) is the *P*-reduction on *Y* of the following causality graph *G'*, which is *P*-consistent on *Y*:

> The vertices of *G'* are the vertices of *G* projected to *P* on *Y*, excluding empty projections. There is an edge between two vertices v_1 and v_2 in *G'* if there is an edge from the *G*-vertex corresponding to v_1 to the *G*-vertex corresponding to v_2.

If *G* is a multi-ledger consistent causality graph, then the *P*-projection on *Y* is *P*-consistent on *Y*, too.

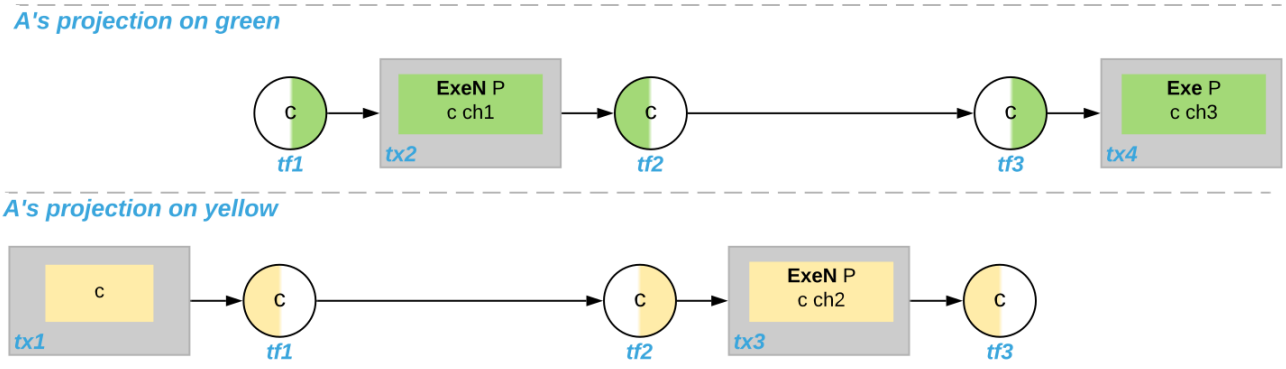
For example, the *multi-ledger causality graph for the split paint counteroffer workflow* is projected as follows:


Alice's projection on green and yellow
CounterOffer $A P Bank
tx2
Exe P (CounterOffer $A P Bank)
Exe A (Iou $Bank A)
PaintAgree $P $A
Iou $Bank P
tx4
Iou $Bank A
tx1
ShowIou $A P Bank
Exe A (ShowIou $A P Bank)
Fetch A (Iou $Bank A)
tx3
Alice's projection on green
Alice's projection on yellow
Bank's projection on yellow
Painter's projection on green and yellow
Painter's projection on green
Painter's projection on yellow

The following points are worth highlighting:

In Alice's projection on the green ledger, Alice witnesses the archival of her *Iou*. As explained in the *Ledger API Ordering Guarantees* below, the **Exercise** action is marked as merely being witnessed in the transaction stream of a Participant Node that hosts Alice on the green ledger but not on the yellow ledger. Similarly, the Painter merely witnesses the **Create** of his *Iou* in the Painter's projection on the green ledger.
In the Painter's projections, the *ShowIou* transaction *tx3* is unordered w.r.t. to the *CounterOffer* acceptance in *tx4* like in the *case of ordinary causality graphs*. The edge *tx3* -> *tx4* is removed by the reduction step during projection.

The projection of transfer actions can be illustrated with the *Multi-Ledger causality graph with transfer actions*. The *A*-projections on the yellow and green ledger look as follows. The white color indicates that a transfer action has no incoming or outgoing ledger annotation. That is, a **Leave** action is white on the right hand side and an **Enter** action is white on the left hand side.



1.47.4 Ledger API Ordering Guarantees

The Transaction Service and the Active Contract Service are derived from the local ledger that the Participant Node maintains for the party. Let *Y* be the set of ledgers on which the Participant Node hosts a party. The transaction tree stream outputs a topological sort of the party's local ledger on *Y*, with the following modifications:

1. **Transfer** actions with either an incoming or an outgoing ledger annotation are output as **Enter** and **Leave** events. **Transfer** actions with both incoming and outgoing ledger annotations are omitted.
2. The incoming and outgoing ledger annotations are not output. Transaction actions with an incoming or outgoing ledger annotation that is not in *Y* are marked as merely being witnessed if the party is an informee of the action.
3. **Fetch** nodes and **NoSuchKey** are omitted.

The flat transaction stream contains precisely the `CreatedEvent`s, `ArchivedEvent`s, and the **Enter** and **Leave** actions that correspond to **Create**, consuming **Exercise**, **Enter** and **Leave** actions in transaction trees on the transaction tree stream where the party is a stakeholder of the affected contract and that are not marked as merely being witnessed.

Similarly, the active contract service provides the set of contracts that are active at the returned offset according to the flat transaction stream. That is, the contract state changes of all events from the transaction event stream are taken into account in the provided set of contracts.

The *ordering guarantees* for single Daml ledgers extend accordingly. In particular, interoperability ensures that all local ledgers are projections of a virtual shared multi-ledger causality graph that con-

nects to the Daml Ledger Model as described above. The ledger validity guarantees therefore extend via the local ledgers to the Ledger API.

1.48 Non-repudiation

The non-repudiation middleware, API and client library are only available in Daml Enterprise and are currently an *Early Access Feature in Alpha status*.

When you are issuing a command over the Ledger API, there is an implicit trust assumption between the issuer of the command and the operator of the participant that the latter will not issue commands on behalf of the former.

The non-repudiation middleware and its client library are a Daml Enterprise exclusive feature that allows ledger operators to run participant nodes that will require each command to come with a verifiable cryptographic signature, which will persisted by the operator. As the sole owner of the private key used to sign the command, the authenticity of the command is thus verified and preserved, ensuring that an operator cannot issue a command on behalf of the user and that the user cannot repudiate the command.

Note that this is an early access feature: its status is currently under development and further feedback can change how certain details might work once the feature is declared a stable part of Daml Enterprise. If you are interested in this feature, you are welcome to use it and give us feedback that will shape how this feature will ultimately come to be.

1.48.1 Architecture

The non-repudiation system consists of three components:

the non-repudiation middleware is a reverse proxy that sits in front of the Ledger API that verifies command signatures and forwards the signed command to the actual participant node
the non-repudiation API is a web server used by the operator to upload new certificates and verify repudiation claims
the non-repudiation client is a gRPC interceptor that can be used alongside any gRPC client on the JVM, including the official Java bindings, that will ensure that commands are signed with a given private key

1.48.2 Run the Server-side Components

The server-side components are the middleware and the API. Both can be run as a single process by running the non-repudiation fat JAR provided as part of Daml Enterprise.

Note that at the current stage you need to also have a PostgreSQL server running where signed commands will be persisted.

The following example shows how to run the non-repudiation server components by connecting to a participant at localhost:6865 and proxying it to the 6866 port, using the given PostgreSQL instance to persist signed commands and certificates.

```
java -jar /path/to/the/non-repudiation.jar --ledger-host localhost --ledger-port
↪6865 --proxy-port 6866 --jdbc url=jdbc:postgresql:nr,user=nr,password=nr
```

For details on how to run them, please run the fat JAR with the `--help` command line option.

1.48.3 Use the Client

The client is a gRPC interceptor which is available to Daml Enterprise users (hence, it's not available on Maven Central).

The Maven coordinates for the library are *com.daml:non-repudiation-client*.

The following example shows how to use this interceptor with the official Java bindings

```
PrivateKey key = readYourPrivateKey();
X509Certificate certificate = readYourX509Certificate();
NettyChannelBuilder builder = NettyChannelBuilder.forAddress(hostname, port);
builder.intercept(SigningInterceptor.signCommands(key, certificate));
DamlLedgerClient client = DamlLedgerClient.newBuilder(builder).build();
client.connect();
```

1.48.4 Non-repudiation Over the HTTP JSON API

The non-repudiation middleware acts *exclusively* as a reverse proxy in front of the Ledger API server: if you want to use the HTTP JSON API you will need to run your own HTTP JSON API server and start it with a certificate that will be used to sign every command issued by the HTTP JSON API to the participant.

The HTTP JSON API bundled with Daml Enterprise has the following extra command line options that *must* be used to run an HTTP JSON API server against the non-repudiation middleware:

-non-repudiation-certificate-path: the path to the X.509 certificate containing the public counterpart to the private key that will be used to sign the commands
-non-repudiation-private-key-path: the path to the file containing the private key that will be used to sign the commands
-non-repudiation-private-key-algorithm: the name of the cryptographic algorithm of the private key (for a list of names supported in the OpenJDK: https://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html#KeyFactory)

1.48.5 TLS Support

At the current stage the non-repudiation feature does not support running against secure Ledger API servers. This will be added as part of stabilizing this feature.

Bibliography

[Cit1] Jones, S. Peyton, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering. ACM SIG-PLAN Notices 35.9 (2000): 280-292.

[Cit2] Jones, SL Peyton, and J. M. Eber. How to write a financial contract , volume Fun Of Programming of Cornerstones of Computing. (2005).