Daml SDK Documentation

Digital Asset

Version : 2.0.0

Copyright (c) 2022 Digital Asset (Switzerland) GmbH and/or its affiliates. All rights reserved. Any unauthorized use, duplication or distribution is strictly prohibited.

Table of contents

Table of contents i

1 Getting started **1**
- 1.1 Installing the SDK . . . 1
  - 1.1.1 1. Installing the Dependencies . . . 1
  - 1.1.2 2. Installing the SDK . . . 1
  - 1.1.3 Installing Daml Enterprise . . . 2
  - 1.1.4 Downloading Manually . . . 2
  - 1.1.5 Next Steps . . . 2
- 1.2 Getting Started with Daml . . . 6
  - 1.2.1 Prerequisites . . . 7
  - 1.2.2 Running the App . . . 7
- 1.3 App Architecture . . . 11
  - 1.3.1 The Daml Model . . . 12
  - 1.3.2 TypeScript Code Generation . . . 14
  - 1.3.3 The UI . . . 14
- 1.4 Your First Feature . . . 17
  - 1.4.1 Daml Changes . . . 17
  - 1.4.2 Messaging UI . . . 18
  - 1.4.3 Running the Updated UI . . . 22
  - 1.4.4 Next Steps . . . 24
- 1.5 Testing Your Web App . . . 24
  - 1.5.1 Setting up our tests . . . 24
  - 1.5.2 Example: Logging in and out . . . 25
  - 1.5.3 Accessing UI elements . . . 26
  - 1.5.4 Writing CSS Selectors . . . 27
  - 1.5.5 The Full Test Suite . . . 28

2 Daml Guide **37**
- 2.1 Writing Daml . . . 37
  - 2.1.1 An introduction to Daml . . . 37
  - 2.1.2 Language reference docs . . . 122
  - 2.1.3 The standard library . . . 169
  - 2.1.4 Good design patterns . . . 243
- 2.2 Building Applications . . . 260
  - 2.2.1 Application architecture . . . 260
  - 2.2.2 JavaScript Client Libraries . . . 267
  - 2.2.3 HTTP JSON API Service . . . 272
  - 2.2.4 Daml Script . . . 327
  - 2.2.5 Daml REPL . . . 341

2.2.6 Upgrading and Extending Daml applications . . . 343
2.2.7 Authorization . . . 354
2.2.8 The Ledger API . . . 358
2.2.9 Command deduplication . . . 494
2.2.10 Daml Triggers - Off-Ledger Automation in Daml . . . 501
2.2.11 Trigger Service . . . 521
2.2.12 Auth Middleware . . . 534
2.3 Overview of Daml ledgers . . . 542
2.3.1 Deploying to a generic Daml ledger . . . 543
2.4 Operating Daml . . . 544
2.4.1 Participant Pruning . . . 544
2.4.2 Participant Metering . . . 547
2.4.3 System Requirements . . . 548
2.5 Developer Tools . . . 549
2.5.1 Daml Assistant (`daml`) . . . 549
2.5.2 Daml Studio . . . 554
2.5.3 Daml Sandbox . . . 564
2.5.4 Navigator . . . 576
2.5.5 Daml codegen . . . 586
2.5.6 Daml Profiler . . . 588

3 Canton Guide **591**
3.1 Introduction to Canton . . . 591
3.2 Tutorials . . . 591
3.2.1 Canton Demo . . . 593
3.2.2 Getting Started . . . 593
3.2.3 Daml SDK and Canton . . . 612
3.2.4 Composability . . . 615
3.3 User Manual . . . 631
3.3.1 Installing Canton . . . 631
3.3.2 Running in Docker . . . 637
3.3.3 Static Configuration . . . 639
3.3.4 Canton Administration APIs . . . 652
3.3.5 Command-line Arguments . . . 688
3.3.6 Canton Console . . . 691
3.3.7 Contract Keys in Canton . . . 779
3.3.8 Enterprise Drivers . . . 789
3.3.9 Error codes . . . 798
3.3.10 High Availability Usage . . . 848
3.3.11 Identity Management . . . 854
3.3.12 Monitoring . . . 866
3.3.13 Operational Processes . . . 887
3.3.14 Versioning . . . 903
3.3.15 Frequently Asked Questions . . . 905
3.4 Architecture In-Depth . . . 911
3.4.1 High-Level Requirements . . . 911
3.4.2 Overview and Assumptions . . . 927
3.4.3 Domain Architecture and Integrations . . . 945
3.4.4 High Availability . . . 956
3.4.5 Identity Management . . . 963
3.4.6 Research Publications . . . 985

3.4.7 Security . . . 986

4 Help 987
4.1 Troubleshooting . . . 987
4.1.1 Error: <X> is not authorized to commit an update . . . 987
4.1.2 Error Argument is not of serializable type . . . 987
4.1.3 Modeling questions . . . 988
4.1.4 Testing questions . . . 990
4.2 Getting Help . . . 990
4.2.1 Support expectations . . . 991
4.3 Portability, Compatibility, and Support Durations . . . 991
4.3.1 Ledger API Compatibility: Application Portability . . . 992
4.3.2 Driver and Participant Compatibility: Network Upgradeability . . . 993
4.3.3 SDK, Runtime Component, and Library Compatibility: Daml Upgradeability . . . 993
4.3.4 Ledger API Support Duration . . . 994

5 Reference 995
5.1 Glossary of concepts . . . 995
5.1.1 Key Concepts . . . 995
5.1.2 Daml Language Concepts . . . 996
5.1.3 Developer tools . . . 1001
5.1.4 Building applications . . . 1002
5.1.5 Canton Concepts . . . 1005
5.2 Daml Ledger Model . . . 1007
5.2.1 Structure . . . 1008
5.2.2 Integrity . . . 1015
5.2.3 Privacy . . . 1027
5.2.4 Daml: Defining Contract Models Compactly . . . 1036
5.2.5 Exceptions . . . 1037
5.3 Identity and Package Management . . . 1044
5.3.1 Identity Management . . . 1045
5.3.2 Package Management . . . 1046
5.4 Time . . . 1048
5.4.1 Ledger time . . . 1048
5.4.2 Record time . . . 1048
5.4.3 Guarantees . . . 1048
5.4.4 Ledger time model . . . 1048
5.4.5 Assigning ledger time . . . 1049
5.5 Causality and Local Ledgers . . . 1049
5.5.1 Causality examples . . . 1050
5.5.2 Causality graphs . . . 1053
5.5.3 Local ledgers . . . 1057
5.6 Daml Ecosystem Overview . . . 1060
5.6.1 Status Definitions . . . 1060
5.6.2 Feature and Component Statuses . . . 1063
5.6.3 Architecture . . . 1066
5.6.4 Daml Networks . . . 1066
5.6.5 Participant Nodes . . . 1067
5.6.6 Ledger API . . . 1067
5.6.7 Daml Components . . . 1068
5.7 Releases and Versioning . . . 1068
5.7.1 Versioning . . . 1068

5.7.2 Cadence . . . 1069
5.7.3 Support Duration . . . 1069
5.7.4 Release Notes . . . 1069
5.7.5 Roadmap . . . 1069
5.7.6 Process . . . 1069

6 Early Access **1071**
6.1 Ledger Export . . . 1071
6.1.1 Introduction . . . 1071
6.1.2 Usage . . . 1071
6.1.3 Output . . . 1072
6.1.4 Executing the Export . . . 1073
6.1.5 Ledger Offsets . . . 1073
6.1.6 Unknown Contract Ids . . . 1074
6.1.7 Transaction Time . . . 1074
6.1.8 Caveats . . . 1074
6.2 Visualizing Daml Contracts . . . 1074
6.2.1 Example: Visualizing the Quickstart project . . . 1075
6.2.2 Visualizing Daml Contracts - Within IDE . . . 1075
6.2.3 Visualizing Daml Contracts - Interactive Graphs . . . 1075
6.3 Ledger Interoperability . . . 1076
6.3.1 Interoperability examples . . . 1076
6.3.2 Multi-ledger causality graphs . . . 1079
6.3.3 Ledger-aware projection . . . 1083
6.3.4 Ledger API ordering guarantees . . . 1087
6.4 Non-repudiation . . . 1088
6.4.1 Architecture . . . 1088
6.4.2 Running the server-side components . . . 1088
6.4.3 Using the client . . . 1089
6.4.4 Non-repudiation over the HTTP JSON API . . . 1089
6.4.5 TLS support . . . 1089
6.5 Operating Daml . . . 1089
6.5.1 Daml Helm Chart . . . 1089
6.5.2 Setting Up Auth0 . . . 1106

Chapter 1

Getting started

1.1 Installing the SDK

1.1.1 1. Installing 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*.

1.1.2 2. Installing the SDK

1.1.2.1 Windows 10

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

1.1.2.2 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
manually. You can find instructions on how to do this *here*.

1.1.3 Installing 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`.

1.1.4 Downloading 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*.

1.1.5 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.

1.1.5.1 Setting JAVA_HOME and PATH Variables

Windows

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

Setting 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.

Setting the PATH Variable

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

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 exsting 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.

Linux

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

Setting 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
```

Setting 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
```

Verifying 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`.

1.1.5.2 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 https://keys.openpgp.org --search
   ↪4911A8DFE976ACDFA07130DBE8372C0C1C734C51
   ```

   This should come back with a key belonging to `Digital Asset Holdings, LLC <security@digitalasset.com>`, created on 2019-05-16 and expiring on 023-04-18. 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 E8372C0C1C734C51
   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: 4911 A8DF E976 ACDF A071  30DB E837 2C0C 1C73 4C51
   ```

   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-----

mQENBFzdsasBCADO+ZcfZQATP6ceTh4WfXiL2Z2tetvUZGfTaEs/UfBoJPmQ53bN
90MxudKhgB2mi8DuifYnHfLCvkxSgzfhj2IogV1S+Fa2x99Y819GausJoYfK9gwc
8YWKEkM81F15jA5UWJTsssKNxUddr/sxJIHIFfqGRQ0e6YeAcc5bOAogBE8UrmxE
uGfOt9/MvLpDewjDE+2lQOFi9RZuy7S8RMJLTiq2JWbO5yI50oFKeMQy/AJPmV7y
qAyYUIeZZxvrYeBWi5JDsZ2HOSJPqV7ttD2MvkyXcJCW/Xf8FcleAoWJU09RwVww
BhZSDz+9mipwZBHENILMuVyEygG5A+vc/YptABEBAAG0N0RpZ2l0YWwgQXNzZXQg
SG9sZGluZ3MsIExMQyA8c2VjdXJpdHlAZGlnaXRhbGFzc2V0LmNvbT6JAVQEEwEI
AD4CGwMFCwkIBwIGFQoJCAsCBBYCAwECHgECF4AWIQRJEajf6Xas36BxMNvoNywM
HHNMUQUCYHxZ3AUJB2EPMAAKCRDoNywMHHNMUfJpB/9Gj7Kce6qtrXj4f54eLOf1
RpKYUnBcBWjmrnj8eS9AYLy7C1nkpP4H8OAlDJWxslnY6MjMOYmPNgGzf4/MONxa
PuFbRdfyblkUfujXikI2GFXwyUDEp9J0WOTC9LmZkRxf92bFxTy9rD+Lx9EeBPdi
nfyID2TOKH0fY0pawqjjvnLyVb/WfNUogkhLRpDXFWrykCWDaWQmFgDkLU2nYkb+
YyEfWq4cgF3Sbsa43AToRUpU16rldPwClmtDPS8Ba/SxvcU3l+9ksdcTsIko8BEy
Bw0K5xkRenEDDwpZvTA2bHLs3iBWW6WC52wyUOLzar+ha/YRgNjb8YBlkYbLbwaN
uQENBFzdsasBCAC5fr5pqxFm+AWPc7wiBSt7uKNdxiRJYydeoPqgmYZTvc8Um8pI
6JHtUrNxnx4WWKtj6iSPn5pSUrJbue4NAUsBF5O9LZ0fcQKb5diZLGHKtOZttCaj
Iryp1Rm961skmPmi3yYaHXq4GC/05Ra/bo3C+ZByv/W0JzntOxA3Pvc3c8Pw5sBm
63xu7iRrnJBtyFGD+MuAZxbN8dwYX0OcmwuSFGxf/wa+aB8b7Ut9RP76sbDvFaXx
Ef314k8AwxUvlv+ozdNWmEBxp1wR/Fra9i8EbC0V6EkCcModRhjbaNSPIbgkC0ka
2cgYp1UDgf9FrKvkuir70dg75qSrPRwvFghrABEBAAGJATwEGAEIACYCGwwWIQRJ
Eajf6Xas36BxMNvoNywMHHNMUQUCYHxZ3AUJB2EPMQAKCRDoNywMHHNMUYXRB/0b
Ln55mfnhJUFwaL49Le5I74EoL4vCAya6aDDVx/C7PJlVfr+cXZi9gNJn9RTAjCz3
4yQeg3AFhqvTK/bEH7RvAfqeUf8TqPjI/qDacSFDhZjdsg3GMDolXp0oubp9mN+Y
JFowLzulJ7DXFVyICozuWeixcjtKzlePX0GW80kcPzXCNwukcMrwCf45+OzF6YMb
yA2FyBmjjgAlHKM/oUapVoD2hmO3ptC5CAkfslxrsIUAfoStez9MrGoX1JOCu4qm
aODLV3Mlty4HhdtO2o+Akh6ay5fnrXQ5r2kGa1ICrfoFFKs7oWpSDbsTsgQKexFC
rLmmBKjG6RQfWJyVSUc8
=pVlb
-----END PGP PUBLIC KEY BLOCK-----
EOF

)
```

1.2 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!

1.2.1 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`. You need `node --version` to report at least `14.8.3`; if you have an older version, see this link for additional installation options.
> A terminal application for command line interaction.

1.2.2 Running 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:

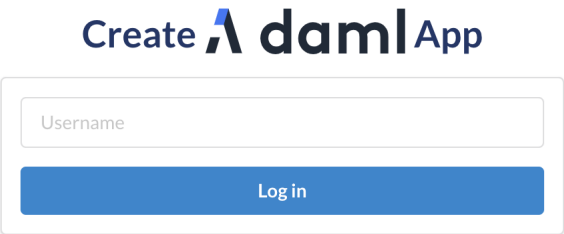
```
npm start
```

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*.

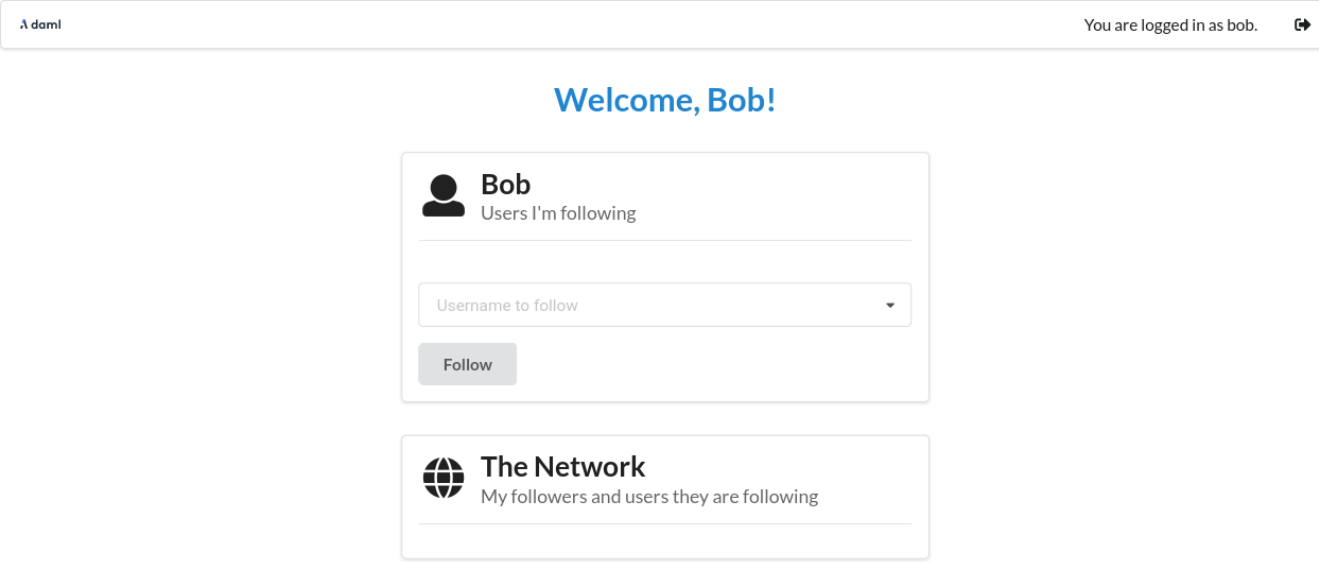


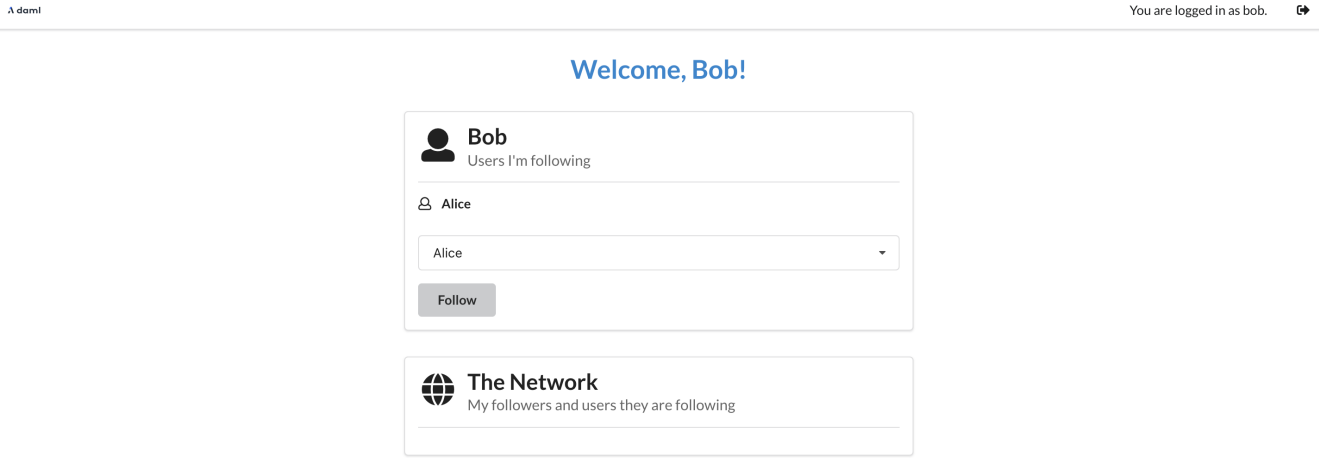
You should see the main screen with two panels. The top panel displays the social network users you are following; the bottom displays the aliases of the users who follow you. Initially these are both empty as you are not following anyone and you don't have any followers. To start following a user, select 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.)

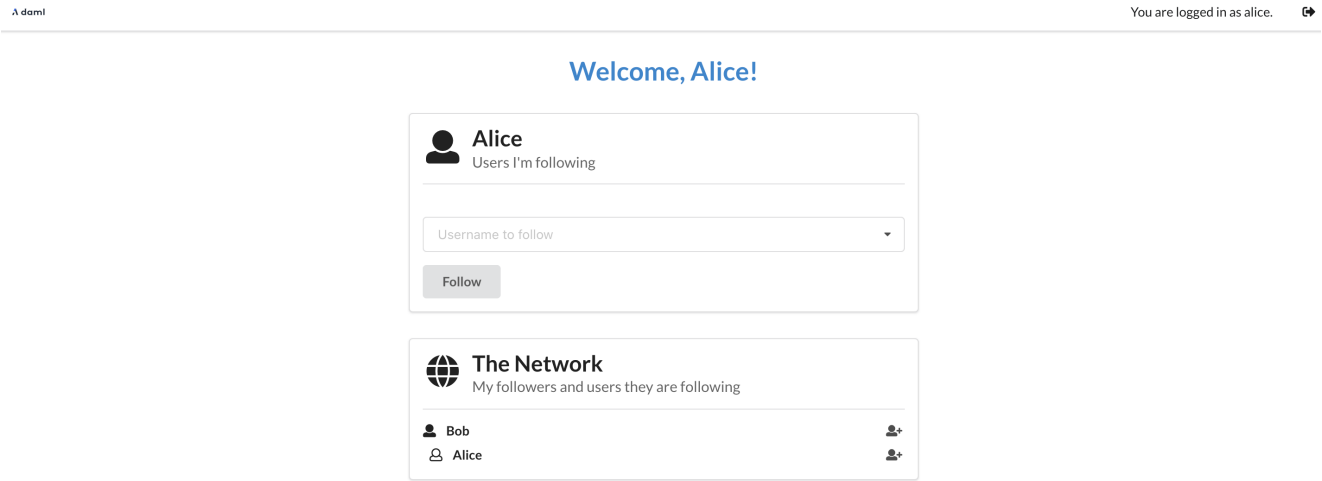
Now that the other user (Alice in this example) has logged in, go back to the previous window/tab, select them drop-down list and click the *Follow* button in the top panel.

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.

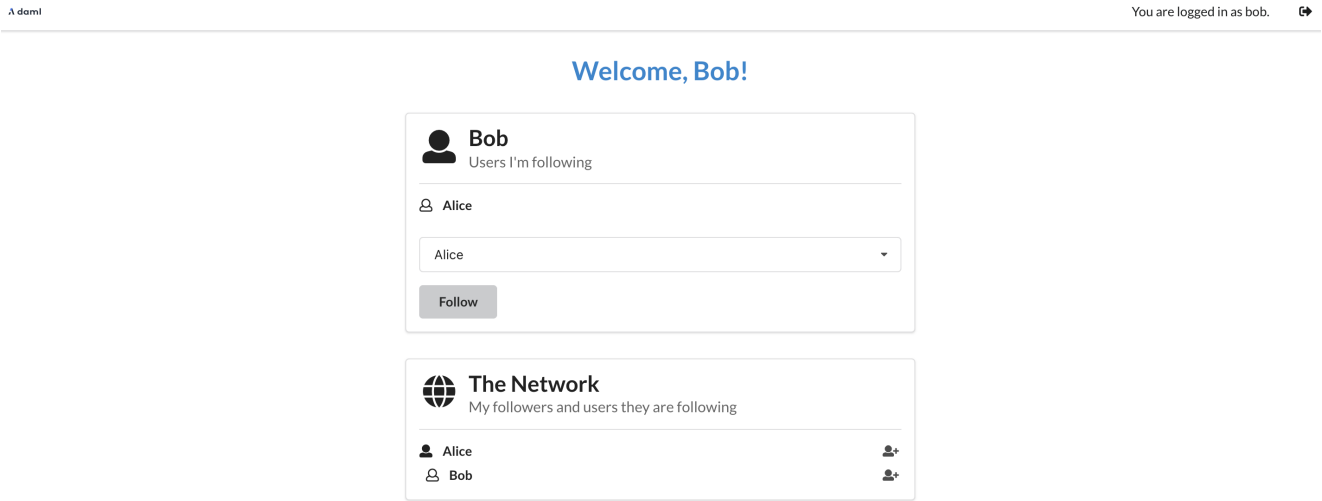

daml
You are logged in as bob.
Welcome, Bob!
Bob
Users I'm following
Username to follow
Follow
The Network
My followers and users they are following


daml
You are logged in as bob.
Welcome, Bob!
Bob
Users I'm following
Alice
Alice
Follow
The Network
My followers and users they are following

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*.

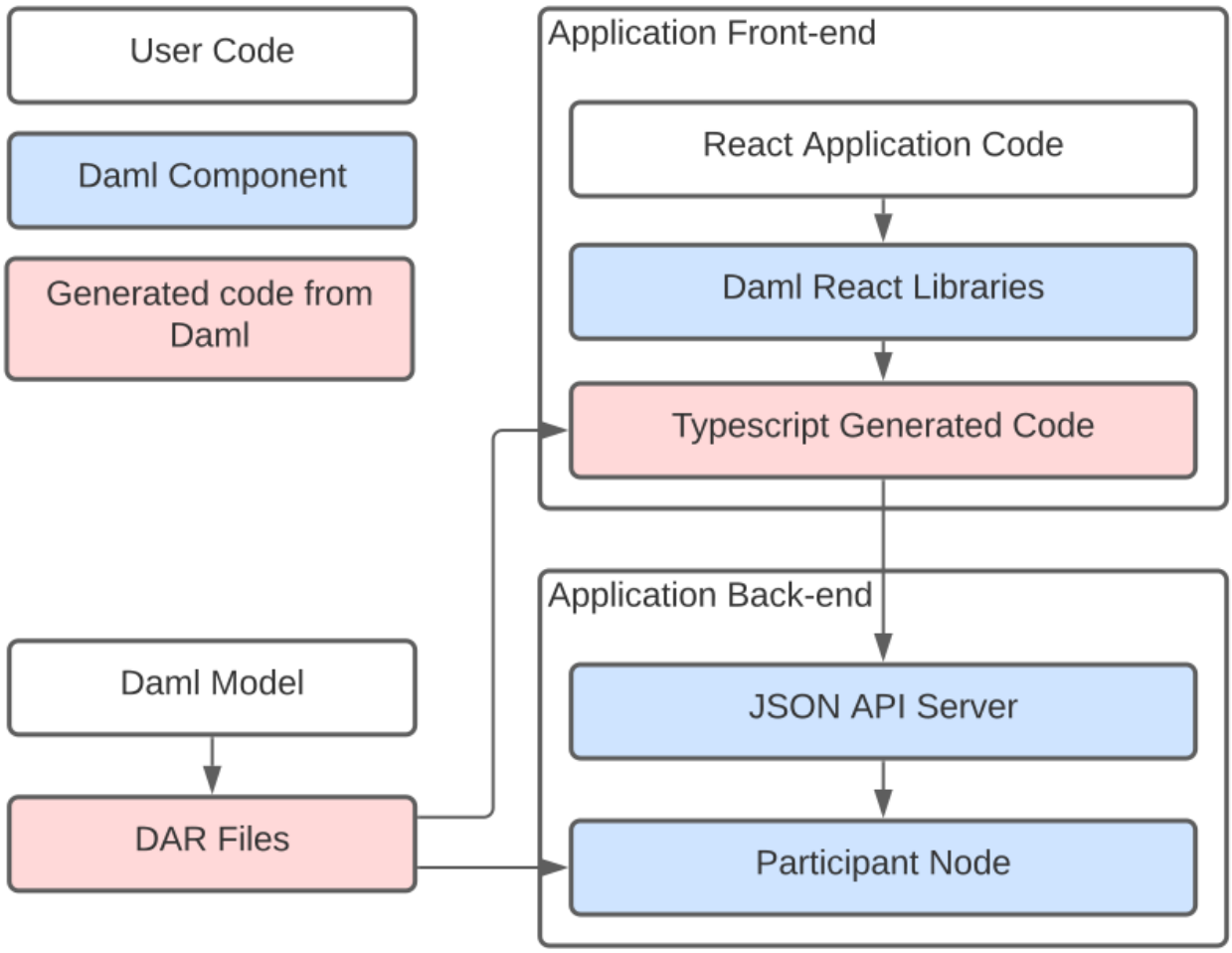
1.3 App Architecture

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 is following the *recommended architecture of a fullstack Daml application*. Below you can see a simplified version of the architecture represented in the app.



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.

1.3.1 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.

1.3.2 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.

1.3.3 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.

1.4 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 four 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.

1.4.1 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.

1.4.2 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`.

1.4.2.1 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.

1.4.2.2 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.

1.4.2.3 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>
```

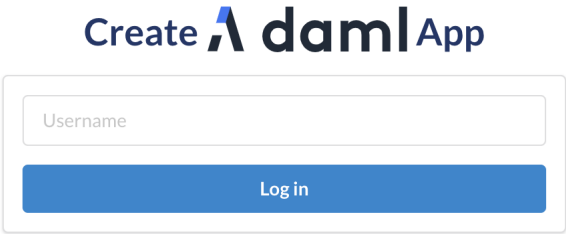
Following the formatting of the previous panels, we include the new messaging components: `MessageEdit` supplied with the usernames of all visible parties as props, and `MessageList` to display all messages.

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

1.4.3 Running 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.



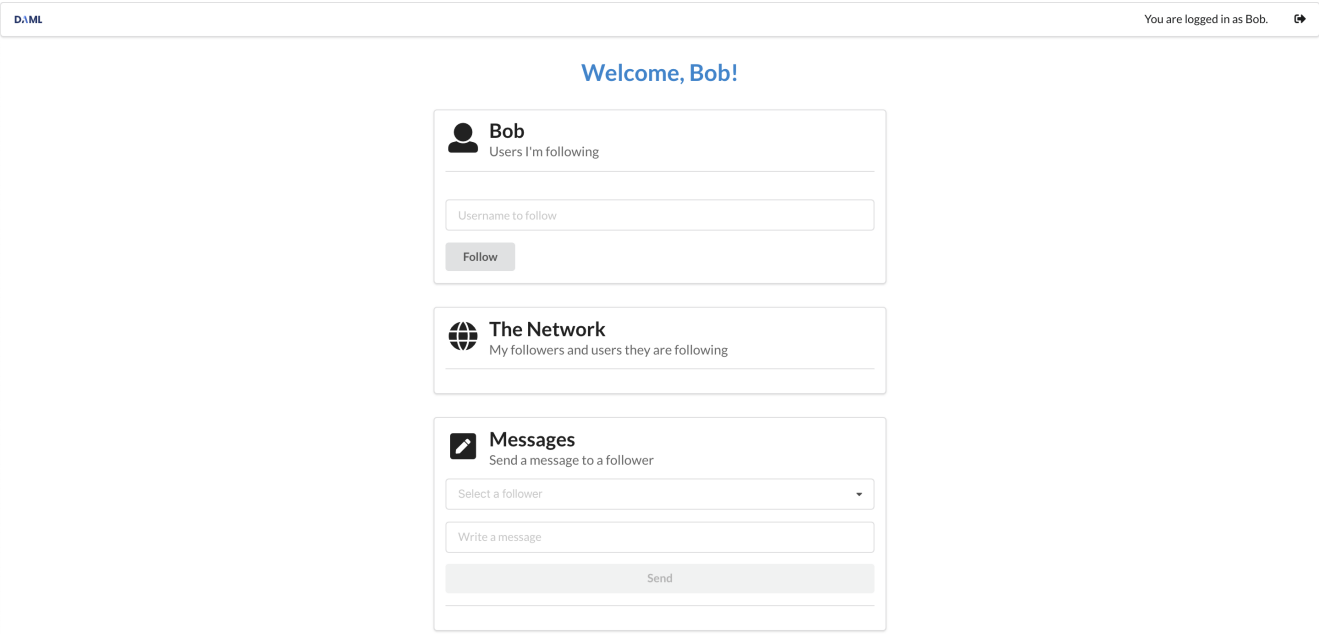
Once you've logged in, you'll see a familiar UI but with our new *Messages* panel at the bottom!

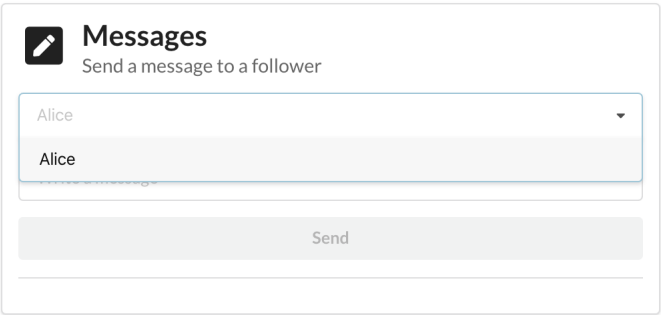
Go ahead and follow more users, and log in as some of those users in separate browser windows to follow yourself back. Then click on the dropdown menu in the *Messages* panel to see a choice of followers to message!

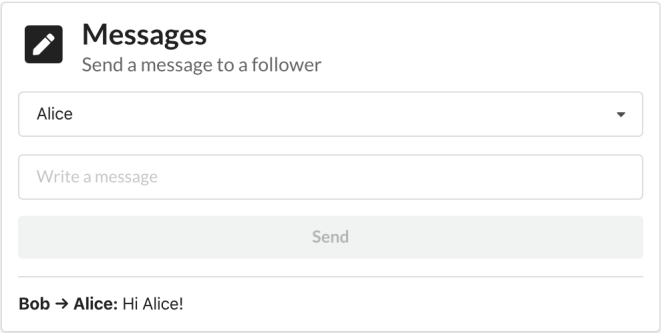
Send some messages between users and make sure you can see each one from the other side. Notice that each new message appears in the UI as soon as it is sent (due to the *streaming* React hooks).

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!

1.4.4 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*.

1.5 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 add --only=dev puppeteer wait-on @types/jest @types/node @types/puppeteer
↪@types/wait-on
```

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.

1.5.1 Setting up our 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.

1.5.2 Example: Logging 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 = authConfig.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.

1.5.3 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",
  );
  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
          primary
          fluid
```

(continues on next page)

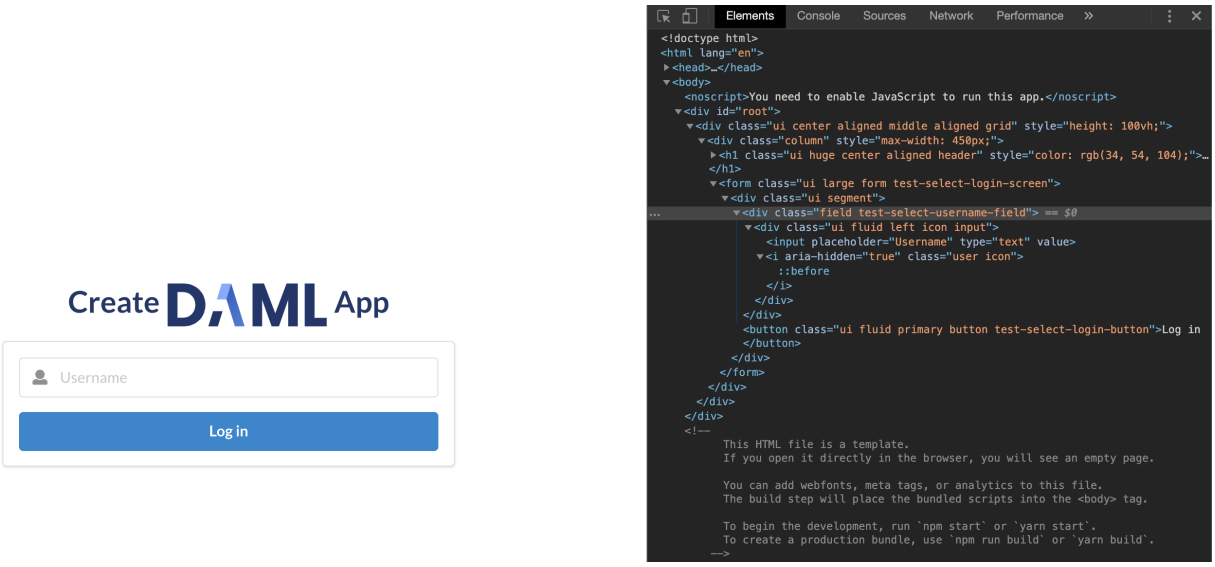
(continued from previous page)

```
          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.

1.5.4 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.

1.5.5 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 } from "@daml/ledger";
import { User } from "@daml.js/create-daml-app";
import { authConfig } 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: authConfig.makeToken("participant_admin"),
});

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 (): [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 = [];
  for (let i = 0; i < 10; ++i) {
```

(continues on next page)

(continued from previous page)

```
    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 () => {
  // If the JSON API server was previously shut down abruptly then the port file
  // may not have been removed.
  // Since we use this file to know when the server is up, we remove it first
  // (if it exists) to be sure.
  const jsonApiPortFilePath = `../${JSON_API_PORT_FILE_NAME}`; // relative to ui
↪folder
  await removeFile(jsonApiPortFilePath);

  // 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" };

  // Arguments for `daml start` (besides those in the `daml.yaml`).
  // The JSON API `--port-file` gives us a file we can check to know that both
  // the sandbox and JSON API server are up and running.
  // We use the default ports for the sandbox and JSON API as done in the
  // Getting Started Guide.
  const startArgs = [
    "start",
    `--json-api-option=--port-file=${JSON_API_PORT_FILE_NAME}`,
  ];

  console.debug("Starting daml start");

  startProc = spawn("daml", startArgs, startOpts);

  await waitOn({ resources: [`file:${jsonApiPortFilePath}`] });

  console.debug("daml start API are 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
```

(continues on next page)

(continued from previous page)

```
  const env = { ...process.env, BROWSER: "none" };
  console.debug("Starting npm start");
  uiProc = spawn("npm-cli.js", ["run-script", "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:localhost:${UI_PORT}`] });
  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) {
    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 = authConfig.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);
```

(continues on next page)

(continued from previous page)

```
  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 =>
    console.log(
      `${message.type().substr(0, 3).toUpperCase()} ${message.text()}`,
    ),
  );
  await page.goto(`http://localhost:${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 => 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",
  );
  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");
};
```

(continues on next page)

(continued from previous page)

```
// 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");
  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.
  // (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 = authConfig.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();
```

(continues on next page)

(continued from previous page)

```
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);

// 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);
```

(continues on next page)

(continued from previous page)

```
  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.
  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);
```

(continues on next page)

(continued from previous page)

```
  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();

  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",
  );
  await usernameInput.click();
  await usernameInput.type(partyName);
  await page.click(".test-select-login-button");
  await page.waitForFunction(
    async () => (await window.getError()) !== undefined,
  );
  expect(dismissError).toHaveBeenCalled();
  return error;
};

test("error on user id with invalid format", async () => {
  // user ids must be lowercase
  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 () => {
```

(continues on next page)

(continued from previous page)

```
  const invalidUser = "nonexistent";
  const page = await newUiPage();
  const error = await failedLogin(page, invalidUser);
  expect(error).toMatch(
    /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);
```

Chapter 2

Daml Guide

2.1 Writing Daml

2.1.1 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 `1_Token`, run `daml new 1_Token --template daml-intro-1`.

Prerequisites:

> You have installed the *Daml SDK*

Next: *1 Basic contracts*.

2.1.1.1 1 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 `1_Token` by running `daml new 1_Token --template daml-intro-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*.

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
```

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.

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.

Next up

In *2 Testing templates using Daml Script*, you'll learn about how to try out the `Token` contract template in Daml's inbuilt Daml Script testing language.

2.1.1.2 2 Testing templates using Daml Script

In this section you will test the `Token` model from *1 Basic contracts* using the *Daml Script* integration in *Daml Studio*. You'll learn about the basic features of :

- Allocating parties
- Submitting transactions
- Creating contracts
- Testing for failure
- Archiving contracts
- Viewing ledger and final ledger state

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

Script basics

A `Script` is like a recipe for a test, where you can script different parties submitting a series of transactions, to check that your templates behave as you'd 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 that is `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 the script is running on, `allocateParty` will either give you back a party with the name you specified or append a suffix to that name if such a party has already been allocated.
> More on `Actions` and `do` blocks in *5 Adding 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. Unsurprisingly, you do this 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 that will be submitted to the ledger creating a transaction which creates a `Token` with owner Alice.

You'll learn all about the syntax `Token with owner = alice` in *3 Data types*.

You could write this as `submit alice (createCmd Token with owner = alice)`, but just like 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.

Running scripts

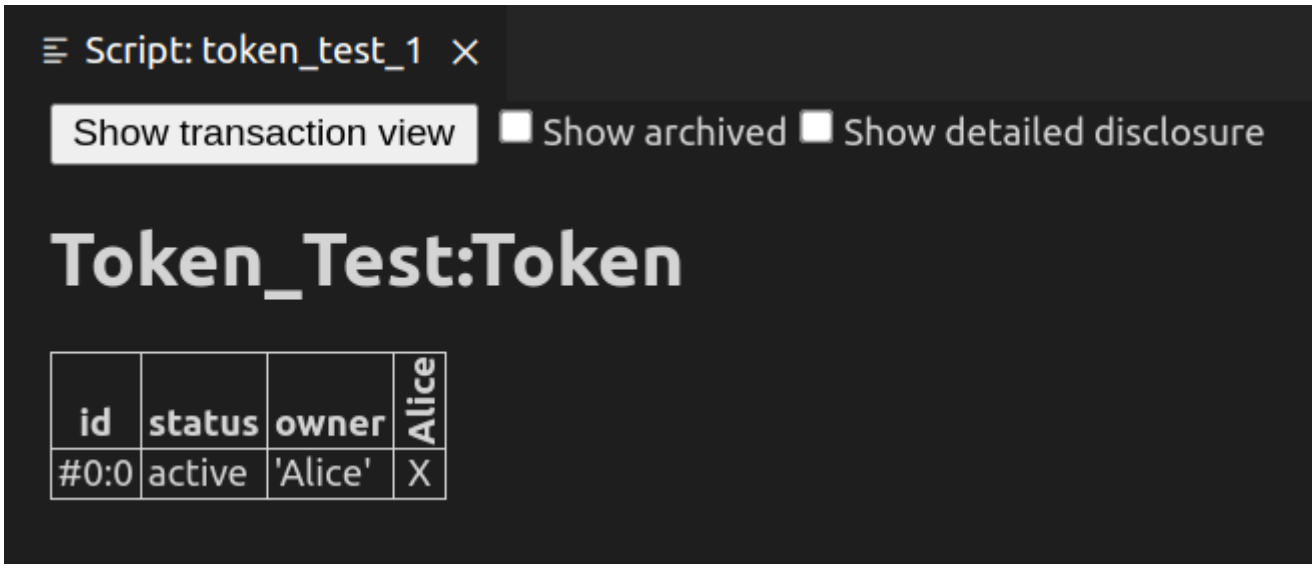
There are a few ways to run Daml Scripts:

> 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, take a look at 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 it to display the outcome of the script.

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

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` is the identifier of the type of contract that's listed below. `Token_Test` is the module name, `Token` 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, 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 damlc -- test --files Token_Test.daml`. If your file contains more than one script, all of them will be run.

Testing for failure

In *1 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 and 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:

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

```
Scenario execution failed on commit at Token_Test:64:3:
  0: create of Token_Test:Token at DA.Internal.Prelude:381:26
     failed due to a missing authorization from 'Bob'

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

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

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 just 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.

Archiving contracts

Archiving contracts works just like creating them, but using `archiveCmd` instead of `createCmd`. Where `createCmd` takes an instance of a template, `archiveCmd` takes a reference to a contract.

References to contracts have the type `ContractId a`, where `a` is a *type parameter* representing the type of 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 to get a handle on its contract ID. In scripts, you do this using `<-` notation. That's because the contract ID needs to be retrieved from the ledger. How this works is discussed in *5 Adding constraints to a contract*.

This script first checks that Bob cannot archive Alice's Token and then Alice successfully archives it:

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

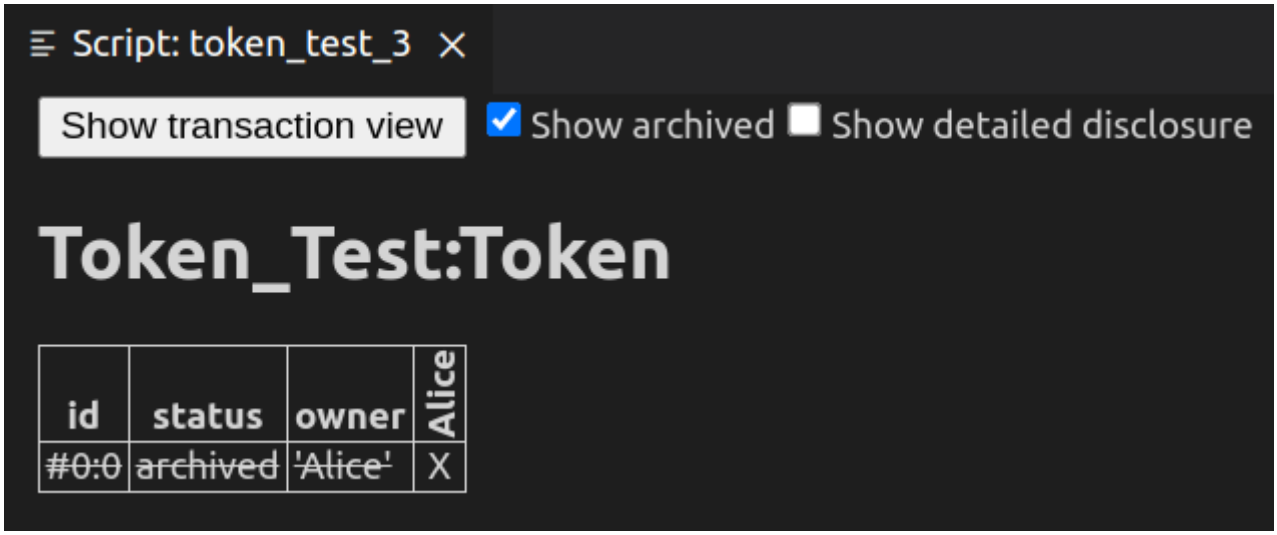
(continues on next page)

(continued from previous page)

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

submitMustFail bob do
  archiveCmd alice_token

submit alice do
  archiveCmd alice_token
```

Exploring the ledger

The resulting script view is empty, because there are no contracts left on the ledger. However, 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. The lines starting with `TX` indicate that there are three committed transactions, with ids `#0`, `#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.

```
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
  │   known to (since): 'Alice' (0)
  └─> create 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
  │   known to (since): 'Alice' (2)
  └─> 'Alice' exercises Archive on #0:0 (Token_Test:Token)
              with

Active contracts:

Return value: {}
```

`known to (since): 'Alice' (#0)` tells you who knows about the contract. The fact that `'Alice'` appears in the list is equivalent to a `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.

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 *Archiving 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 *Testing for failure* we saw a proper error message for a failing submit.

Next up

In *3 Data types* you will learn about Daml's type system, and how you can think of templates as tables and contracts as database rows.

2.1.1.3 3 Data types

In *1 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 *2 Testing 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 `3_Data` by running `daml new 3_Data --template daml-intro-3`

Native types

You have already encountered a few native Daml types: `Party` in *1 Basic contracts*, and `Text` and `ContractId` in *2 Testing 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 `-9999999999999999999999999999.9999999999`.
`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

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

Assembling 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
```

(continues on next page)

(continued from previous page)

```
  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

-- 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"
```

(continues on next page)

(continued from previous page)

```
      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
```

(continues on next page)

(continued from previous page)

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

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

cash_balance_test = script do
```

(continues on next page)

(continued from previous page)

```
  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)

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

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

template CashBalance
```

(continues on next page)

(continued from previous page)

```
  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
```

(continues on next page)

(continued from previous page)

```
  | 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
    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
```

(continues on next page)

(continued from previous page)

```
      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
```

Manipulating 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)

  -- 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.

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
    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
```

(continues on next page)

(continued from previous page)

```
      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_account <- submit accountant do
    archiveCmd accountCid
    createCmd account with
      bank = account.bank with
        telephone = "098 7654 321"
    pure ()

  -- 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 *7 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
```

(continues on next page)

(continued from previous page)

```
  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

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

  Some account <- queryContractId accountant accountCid
  balanceCid <- submit accountant do
```

(continues on next page)

(continued from previous page)

```
    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
  (cid, account) <- submit accountant do
    createAndExerciseCmd (Helper accountant) (FetchAccountByKey balance.account)
  assert (cid == new_accountCid)

-- Helper template to call `fetchByKey`.
template Helper
  with
    p : Party
  where
    signatory p
    choice FetchAccountByKey : (ContractId Account, Account)
      with
        accountKey : AccountKey
      controller p
      do fetchByKey @Account accountKey
```

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, we use `fetchByKey @Account`. `fetchByKey @Account` takes a value of type `AccountKey` and returns a tuple `(ContractId Account, Account)` if the lookup was successful or fails the transaction otherwise. `fetchByKey` cannot be used directly in the list of commands sent to the ledger. Therefore we create a `Helper` template with a `FetchAccountByKey` choice and call that via `createAndExerciseCmd`. We will learn more about choices in the *next section*.

Since a single type could be used as the key for multiple templates, you need to tell the compiler what type of contract is being fetched by using the `@Account` notation.

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 *4 Transforming data using choices* you'll learn how to define data transformations and give other parties the right to manipulate data in restricted ways.

2.1.1.4 4 Transforming 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 `4_Transformations` by running `daml new 4_Transformations --template daml-intro-4`

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 *3 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.

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.

Choices in the Ledger Model

In *1 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:43:17)
  #0:0
  │   consumed by: #2:0
  │   referenced by #2:0
  │   known to (since): 'Alice' (#0), 'Bob' (#0)
  └─> create Contact:Contact
      with
        owner = 'Alice'; party = 'Bob'; address = "1 Bobstreet"; telephone = "012
↪345 6789"

  TX #1 1970-01-01T00:00:00Z
    mustFailAt 'Bob' (Contact:52:3)

  TX #2 1970-01-01T00:00:00Z (Contact:56:22)
  #2:0
  │   known to (since): 'Alice' (#2), 'Bob' (#2)
  └─> 'Alice' exercises UpdateTelephone on #0:0 (Contact:Contact)
              with
                newTelephone = "098 7654 321"
      children:
      #2:1
      │   consumed by: #4:0
      │   referenced by #3:0, #4:0
      │   known to (since): 'Alice' (#2), 'Bob' (#2)
      └─> create Contact:Contact
          with
            owner = 'Alice'; party = 'Bob'; address = "1 Bobstreet"; telephone =
↪"098 7654 321"

  TX #3 1970-01-01T00:00:00Z (Contact:60:3)
  #3:0
  └─> fetch #2:1 (Contact:Contact)
```

(continues on next page)

(continued from previous page)

```
  TX #4 1970-01-01T00:00:00Z (Contact:66:22)
  #4:0
  │   known to (since): 'Alice' (#4), 'Bob' (#4)
  └─> 'Bob' exercises UpdateAddress on #2:1 (Contact:Contact)
            with
              newAddress = "1-10 Bobstreet"
      children:
      #4:1
      │   referenced by #5:0
      │   known to (since): 'Alice' (#4), 'Bob' (#4)
      └─> create Contact:Contact
          with
            owner = 'Alice';
            party = 'Bob';
            address = "1-10 Bobstreet";
            telephone = "098 7654 321"

  TX #5 1970-01-01T00:00:00Z (Contact:70:3)
  #5:0
  └─> fetch #4:1 (Contact:Contact)

Active contracts:  #4:1

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 `#4` contain `exercise` actions with IDs `#2:0` and `#4:0`. The `create` actions of the updated, `Contact` contracts, `#2:1` and `#4:1`, 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.

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 *3 Data types* as it's not concerned with the location of the physical cash, but merely with liabilities:

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


module SimpleIou where
```

(continues on next page)

(continued from previous page)

```
import Daml.Script

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

template SimpleIou
  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.

Next up

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

In *5 Adding 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.

2.1.1.5 5 Adding 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 *7 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 `5_Restrictions` by running `daml new 5_Restrictions --template daml-intro-5`

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"
```

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 choice doesn't do anything other than archiving 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
        assertMsg
          ("Cannot redeem outside business hours. Current time: " <> show
↪timeofday)
          (hrs >= 8 && hrs <= 18)
        case dow of
          Saturday -> abort "Cannot redeem on a Saturday."
          Sunday -> abort "Cannot redeem on a Sunday."
          _ -> return ()
```

```
  -- June 1st 2019 is a Saturday.
  setTime (time (date 2019 Jun 1) 0 0 0)
  -- Bob cannot redeem on a Saturday.
```

(continues on next page)

(continued from previous page)

```
  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
```

There are quite a few new time-related functions from the `DA.Time` and `DA.Date` libraries here. Their names should be reasonably descriptive so how they work won't be covered here, but given that Daml assumes it is run in a distributed setting, we will still discuss time in Daml.

There's also quite a lot going on inside the `do` block of the `Redeem` choice, with several uses of the `<-` operator. `do` blocks and `<-` deserve a proper explanation at this point.

Time on Daml ledgers

Each transaction on a Daml ledger has two timestamps called the *ledger time (LT)* and the *record time (RT)*. The ledger time is set by the participant, the record time is set by the ledger.

Each Daml ledger has a policy on the allowed difference between LT and RT called the *skew*. The participant has to take a good guess at what the record time will be. If it's too far off, the transaction will be rejected.

`getTime` is an action that gets the LT from the ledger. In the above example, that 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.

Consider the following example: Suppose that the ledger had a skew of 10 seconds. At 17:59:55, Alice submits a transaction to redeem an Iou. One second later, the transaction is assigned a LT of 17:59:56, but then takes 10 seconds to commit and is recorded on the ledger at 18:00:06. Even though it was committed after business hours, it would be a valid transaction and be committed successfully as `getTime` will return 17:59:56 so `hrs == 17`. Since the RT is 18:00:06, `LT - RT <= 10 seconds` and the transaction won't be rejected.

Time therefore has to be considered slightly fuzzy in Daml, with the fuzziness depending on the skew parameter.

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.

Time on ledgers

On a distributed Daml ledger, there are no guarantees that ledger time or record time 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`:

```
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
```

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
```

Chaining up 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`.

Wrapping 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`.

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.

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.

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
```

(continues on next page)

(continued from previous page)

```
 | exponent > 0 =
   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.

Next up

You can now specify a precise data and data-transformation model for Daml ledgers. In *6 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.

2.1.1.6 6 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 `6_Parties` by running `daml new 6_Parties --template daml-intro-6`

Preventing IOU revocation

The `SimpleIou` contract from *4 Transforming data using choices* and *5 Adding 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.

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
```

(continues on next page)

(continued from previous page)

```
    signatory iou.issuer
    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
```

(continues on next page)

(continued from previous page)

```
    choice IouTransferProposal_Accept
      : 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`.

Note also how `newOwner` is given multiple choices using a single `controller newOwner can` block. 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
```

(continues on next page)

(continued from previous page)

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

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 *7 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`.

The above syntax is an alternative to `controller c can`, which allows for this. Such choices live outside any `controller c can` block. They declared using the `choice` keyword, and have an extra clause `controller c`, which takes the place of `controller c can`, and has access to the choice arguments.

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
    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
```

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:269:3)
#12:0
│   known to (since): 'Bob' (#12), 'Charlie' (#12)
└─> 'Bob' exercises Send_Iou on #10:0 (Parties:IouSender)
          with
            iouCid = #11:3
    children:
    #12:1
    │   known to (since): 'Bob' (#12), 'Charlie' (#12)
    └─> fetch #11:3 (Parties:Iou)

    #12:2
    │   known to (since): 'Bob' (#12), 'Alice' (#12), 'Charlie' (#12)
    └─> 'Bob', 'Charlie' exercises Mutual_Transfer on #11:3 (Parties:Iou)
                         with
                           newOwner = 'Charlie'
        children:
        #12:3
        │   known to (since): 'Charlie' (#12), 'Alice' (#12), 'Bob' (#12)
        └─> create Parties:Iou
            with
              issuer = 'Alice';
              owner = 'Charlie';
              cash =
                (Parties:Cash with
                   currency = "USD"; amount = 100.0)
```

Note that authority is not automatically transferred transitively.

```
template NonTransitive
  with
    partyA : Party
    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.

Next up

In *7 Composing choices* you will put everything you have learned together to build a simple asset holding and trading model akin to that in the *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.

2.1.1.7 7 Composing choices

It's time to put everything you've learnt 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 *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 `7_Composing` by running `daml new 7Composing --template daml-intro-7`

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 chapter 7 project:

```
sdk-version: __VERSION__
name: __PROJECT_NAME__
source: daml
version: 1.0.0
dependencies:
  - daml-prim
  - daml-stdlib
  - daml-script
sandbox-options:
  - --wall-clock-time
```

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 *9 Working with Dependencies*.

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
```

Project overview

The project both changes and adds to the `Iou` model presented in *6 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.

Composed choices and scripts

This project showcases how you can put the `Update` and `Script` actions you learnt about in *6 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)
```

(continues on next page)

(continued from previous page)

```
        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 #15 1970-01-01T00:00:00Z (Test.Intro.Asset.Trade:77:23)
#15:0
│   known to (since): 'Alice' (#15), 'Bob' (#15)
└─> 'Bob' exercises Trade_Settle on #13:1 (Intro.Asset.Trade:Trade)
          with
            quoteAssetCid = #10:1; baseApprovalCid = #14:2
    children:
    #15:1
```

(continues on next page)

(continued from previous page)

```
    │   known to (since): 'Alice' (#15), 'Bob' (#15)
    └─> fetch #11:1 (Intro.Asset:Asset)

    #15:2
    │   known to (since): 'Alice' (#15), 'Bob' (#15)
    └─> fetch #10:1 (Intro.Asset:Asset)

    #15:3
    │   known to (since): 'USD_Bank' (#15), 'Bob' (#15), 'Alice' (#15)
    └─> 'Alice',
        'Bob' exercises TransferApproval_Transfer on #14:2 (Intro.
↪Asset:TransferApproval)
              with
                assetCid = #11:1
        children:
        #15:4
        │   known to (since): 'USD_Bank' (#15), 'Bob' (#15), 'Alice' (#15)
        └─> fetch #11:1 (Intro.Asset:Asset)

        #15:5
        │   known to (since): 'Alice' (#15), 'USD_Bank' (#15), 'Bob' (#15)
        └─> 'Alice', 'USD_Bank' exercises Archive on #11:1 (Intro.Asset:Asset)

        #15:6
        │   referenced by #17:0
        │   known to (since): 'Bob' (#15), 'USD_Bank' (#15), 'Alice' (#15)
        └─> create Intro.Asset:Asset
            with
              issuer = 'USD_Bank'; owner = 'Bob'; symbol = "USD"; quantity = 100.
↪0; observers = []

    #15:7
    │   known to (since): 'EUR_Bank' (#15), 'Alice' (#15), 'Bob' (#15)
    └─> 'Bob',
        'Alice' exercises TransferApproval_Transfer on #12:1 (Intro.
↪Asset:TransferApproval)
                with
                  assetCid = #10:1
        children:
        #15:8
        │   known to (since): 'EUR_Bank' (#15), 'Alice' (#15), 'Bob' (#15)
        └─> fetch #10:1 (Intro.Asset:Asset)

        #15:9
        │   known to (since): 'Bob' (#15), 'EUR_Bank' (#15), 'Alice' (#15)
        └─> 'Bob', 'EUR_Bank' exercises Archive on #10:1 (Intro.Asset:Asset)

        #15:10
        │   referenced by #16:0
        │   known to (since): 'Alice' (#15), 'EUR_Bank' (#15), 'Bob' (#15)
        └─> create Intro.Asset:Asset
            with
              issuer = 'EUR_Bank'; owner = 'Alice'; symbol = "EUR"; quantity = 90.
↪0; 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
      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.

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.

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.

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 rule 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.

Next up

In *8 Exception Handling*, we will learn about how errors in your model can be handled in Daml.

2.1.1.8 8 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 *3 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 callsites 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 `8_Exceptions` by running `daml new 8_Exceptions --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 as well as a list of observers that can order items.

```
template Shop
  with
    owner : Party
    bank : Party
    observers : [Party]
  where
    signatory owner
    observer observers
    let price: Decimal = 100.0
```

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` matter what. Next, we try 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
```

(continues on next page)

(continued from previous page)

```
    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) (Transfer owner price)
           pure ()
         catch
           PreconditionFailed _ -> do
             create Iou with
               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*.

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 *9 Working 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.

2.1.1.9 9 Working with Dependencies

The application from Chapter 7 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 feature before going live with an application so it's important to understand way 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 though 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 chapter 7 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 chapter 7, the setup for this chapter is slightly more complex:

1. In a base directory, load the chapter 7 project using `daml new 7Composing --template daml-intro-7`. The directory `7Composing` here is important as it'll be referenced by the other project we are creating.
2. In the same directory, load the chapter 8 project using `daml new 9Dependencies --template daml-intro-9`.

`8Dependencies` contains a new module `Intro.Asset.MultiTrade` and a corresponding test module `Test.Intro.Asset.MultiTrade`.

DAR, DALF, Daml-LF, and the Engine

In *7 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 chapter 8 project to the chapter 7 project so it's time to learn a little more about all this.

Let's have a look inside the DAR file of chapter 7. 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 `7Composing` 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 `7Composing-1.0.0-887056cbb313b94ab9a6caf34f7fe4fbfe19cb0c861e` `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.

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 (7Composing) 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 "887056cbb313b94ab9a6caf34f7fe4fbfe19cb0c861e50d1594c665567ab7625" -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 `7Composing-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.

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 `7Composing` with SDK 1.100.0, and are building `8Dependencies` 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 chapter 8 project includes the chapter 7 that way.

```
  - daml-script
data-dependencies:
  - ../7Composing/assets.dar
```

You'll notice a module `Test.Intro.Asset.TradeSetup`, which is almost a carbon copy of the Chapter 7 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.

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 chapter 7, that would also have saved us from copying the chapter 7

Similarly, we included `Trade` in the same project as `Asset` in chapter 7, 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.

Next up

The `MultiTrade` model has more complex control flow and data handling than previous models. In *10 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 chapter 7 and 8 folders just yet.

2.1.1.10 10 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 chapter 7 and 8 projects set up, and want to look back at the code, please follow the setup instructions in *9 Working 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.

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 present in Haskell. But 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.

Functions

In *3 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:

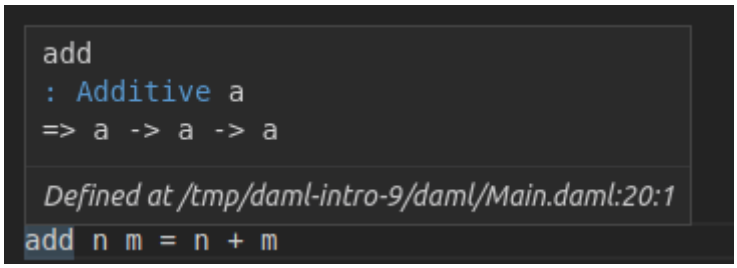
```
increment : Int -> Int
increment n = n + 1
```

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`, ie 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 +)
decrement = (- 1)
```

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 *3 Data types*. The `Additive` typeclass says that you can add a thing. Ie 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 the `+` interface.

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`, map 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.

Pattern Matching in Arguments

You met pattern matching in *3 Data types*, using `case` statements 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
```

Using function pattern matching is clearly the most elegant here. We never need the tuple as a whole, just its members. 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
```

This means `fromSome` is 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 chapter 8:

```
issueAsset : Asset -> Script (ContractId Asset)
issueAsset asset@(Asset with ..) = do
  assetHolders <- queryFilter @AssetHolder issuer
    (\ah -> (ah.issuer == issuer) && (ah.owner == owner))

  case assetHolders of
```

(continues on next page)

(continued from previous page)

```
    (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 name. So the function succinctly transfers all fields except for `owner`, which is set explicitly, from the V1 Asset to the V2 Asset.

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 commonly, it makes sense to define functions locally, inside a `let` clause or similar. A good example of this are the `validate` and `transfer` functions defined locally in the `Trade_Settle` choice of the model from chapter 8:

```
        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, or as choice in- or outputs. They also don't have instances of the `Eq` or `Show` typeclasses which one would commonly want on data types.

You can probably guess what the `mapA` and `mapA_s` in the above choice do. They somehow loop through the lists of assets, and approvals, and the functions `validate` and `transfer` to each, performing the resulting `Update` action in the process. We'll look at that more closely under *Looping* below.

Lambdas

Like in most modern languages, Daml also supports inline functions called lambdas. They are defined using `(\x y z -> ...)` syntax. For example, a lambda version of `increment` would be `(\n -> n + 1)`.

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 Chapter 7 the only real kind of control flow introduced has been `case`, which is a powerful tool for branching.

If..Else

Chapter 5 also showed a seemingly self-explanatory `if..else` statement, but didn't explain it further. And they are actually the same thing. 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` statements are equivalent to a `case` statement, but are easier to read.

Control Flow as Expressions

`case` statements and `if..else` 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 expect. The error only gets evaluated if an invalid text is passed in.

This is different to 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.

But while being proper control flow, `case` and `if..else` statements are also expressions in the sense that they 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` statement, the value of the evaluated function is just the value of the `case`/`if` statement. Things that have a value have a type. The `if..else` expression in `boolToInt2` has type `Int` as that's what the function returns, 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"
```

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
```

(continues on next page)

(continued from previous page)

```
    p : Party
  where
    signatory p
```

It would be tempting to write a simple `if..else`, but it won't typecheck:

```
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 is short circuits evaluation just like `if..else`. `noop` is a no-op, 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 whileFunction(arr) {
  var rev = initialize(input);
  while (doContinue (state)) {
    state = process (state);
  }
  return finalize(state);
}
```

The only real difference is that the iterator is explicit in the former, and implicit in the latter.

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 literally 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)]
```

But you should immediately forget again that you just learnt about `(!!)`. Lists in Daml are linked lists, which makes access using `(!!)` slow and idiosyncratic. The way to do this in Daml is to get rid of the `i` altogether and instead merge the lists first, 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 elements to the two input lists, and so on. It drops any left-over elements of the longer list, thus making the `min` logic unnecessary.

Maps

You've probably noticed that what we've done in this second version of `sumArr` is pretty standard, we have taken a list `zip arr1 arr2` applied a function `\(x, y) -> x + y` to each element, and returned the list of results. This operation is called `map : (a -> b) -> [a] -> [b]`. We can now write `sumArr` even more nicely:

```
sumArrs3 arr1 arr2 = map (\(x, y) -> (x + y)) (zip arr1 arr2)
```

As a rule of thumb: 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 *5 Adding constraints to a contract*: The functions used to map or process items have no side-effects. In day-to-day Daml that's the exception rather than the rule. If you have looked at the chapter 8 models, you'll have noticed `mapA`, `mapA_`, and `forA` all over the place. A good example are the `mapA` 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, ie something of type `[ContractId AssetHolder]`. Using the map function almost gets us there. `map setupRelationship rels : [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 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, of course. `sequence : : Applicative m => [m a] -> m [a]` implements that intuition and allows us to take the `Update` out of the

list. 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 of course, 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.

Have a go at implementing `foldlA` in terms of `foldl` and `sequence` and `mapA` in terms of `foldA`. Here are some 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. `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_`.

Next up

You now know the basics of functions and control flow, both in pure and Action contexts. The Chapter 8 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 *11 Intro to the Daml Standard Library*, you'll get an overview of the library and learn how to search and browse it.

2.1.1.11 11 Intro to the Daml Standard Library

In chapters *3 Data types* and *10 Functional Programming 101* you learnt how to define your own data types and functions. But of course 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, but also learn how to browse and search this 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.

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.

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 chapter 8 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.

Typeclasses

You've seen typeclasses in use all the way from *3 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

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.

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 *3 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 *5 Adding 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 *5 Adding 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`.

Important Modules in the Standard Library

For almost all the types and typeclasses presented above, the Standard Library contains a module:

Module DA.List for Lists
Module DA.Optional for `Optional`
Module DA.Tuple for Tuples
Module DA.Either for `Either`
Module DA.Functor for Functors
Module DA.Action for Actions
Module DA.Monoid and *Module DA.Semigroup* for Monoids and Semigroups
Module DA.Text for working with `Text`
Module DA.Time for working with `Time`
Module 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 learnt about, 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: *Module*

DA.Traversable, and *Module 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.

Searching 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, or even less so to find a function that does a thing you need to do.

Daml has it's own version of the Hoogle search engine, which offers search both by name and by signature. It's 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.

Searching 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.

Searching 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 *10 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.

Next up

There's little more to learn about writing Daml at this point that isn't best learnt by practice and consulting reference material for both Daml and Haskell. To finish off this course, you'll learn a little more about your options for testing and interacting with Daml code in *12 Testing Daml Contracts*, and about the operational semantics of some keywords and common associated failures.

2.1.1.12 12 Testing Daml Contracts

This chapter is all about testing and debugging the Daml contracts you've built using the tools from chapters 1-10. 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
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, please follow the setup instructions in *9 Working with Dependencies* to get hold of the code for this chapter. There is no code specific to this chapter.

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 chapter 8 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.

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 *Module DA.Stack* to do so, but we won't cover that any further here.

Diagnosing 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.

If we look back at *Daml's execution model* we'll see there are three places where ledger state is consumed:

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.
```

Avoiding 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 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.

Next up

You've reached the end of the Introduction to Daml. Congratulations. If you think you understand all this material, you could test yourself by getting Daml certified at https://academy.daml.com. Or put your skills to good use by developing a Daml application. There are plenty of examples to inspire you on the Examples page.

2.1.2 Language reference docs

This section contains a reference to writing templates for Daml contracts. It includes:

2.1.2.1 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 *Contract keys*.

maintainers `maintainer` keyword
: Required if you have specified a `key`. Keys are only unique to a `maintainer`. See *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`

2.1.2.2 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]`).

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.

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
```

(continues on next page)

(continued from previous page)

```
    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.

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 *Contract keys*.

2.1.2.3 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

```
-- 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 (currently
↪only available in Daml-LF 1.11)
      controller exampleParty
        do
          return () -- replace this line with the choice body
```

Choice name

Listing 2: Option 1 for specifying choices: choice name first

```
    choice ExampleChoice1
          : () -- replace () with the actual return type
```

Listing 3: 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 your project. Choices in different templates can't have the same 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 4: Option 1 for specifying choices: choice name first

```
      controller exampleParty
```

Listing 5: 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 6: Option 1 for specifying choices: choice name first

```
preconsuming choice ExampleChoice5
      : () -- replace () with the actual return type
```

Listing 7: 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 8: Option 1 for specifying choices: choice name first

```
postconsuming choice ExampleChoice6
      : () -- replace () with the actual return type
```

Listing 9: 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 10: Option 1 for specifying choices: choice name first

```
nonconsuming choice ExampleChoice3
      : () -- replace () with the actual return type
```

Listing 11: 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.

2.1.2.4 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.

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.
Exercises the specified choice on the specified contract.
Use `with` to specify the choice parameters.
Requires authorization from the controller(s) of the choice **and** from at least one of the maintainers of the key. If the authorization is not given, the transaction fails.

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.
The same as `fetch`, but fetches the contract with that *contract key*, instead of the contract ID. Like `fetch`, `fetchByKey` needs to be authorized by at least one stakeholder of the contract. Fails if no contract can be found.

lookupByKey

```
fetchedContractId <- lookupByKey @ContractType contractKey
```

`lookupByKey` function.
Use this to confirm that a contract with the given *contract key* exists.
If the submitting party is a stakeholder of a matching contract, `lookupByKey` returns the `ContractId` of the contract; otherwise, it returns `None`. Transactions may fail due to contention because the key changes between the lookup and committing the transaction, or becasue the submitter didn't know about the existence of a matching contract.
All of the maintainers of the key must authorize the lookup (by either being signatories or by submitting the command to lookup).

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.

2.1.2.5 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` decimal digits. The scale parameter `n` controls the number of digits after the decimal point, so for example, `Numeric 10` values have 10 digits after the decimal point, and `Numeric 20` values have 20 digits after the decimal point. 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`	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. 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)`	`seconds 1` and `seconds (-2)` represent the values for 1 and -2 seconds. There are no literals for `RelTime`. Instead they are created using one of `days`, `hours`, `minutes` and `seconds` (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 :: [] )
```

Summing a list

To sum a list, use a *fold* (because there are no loops in Daml). See *Folding* 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.

Accessing 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
```

Updating 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.

2.1.2.6 Reference: built-in functions

This page gives reference information on functions for.

Working 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.

Working 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`.

Working 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

Working with lists

Daml has these built-in functions for working with lists:

`foldl` and `foldr`: see *Folding* below.

Folding

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)
```

2.1.2.7 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
```

2.1.2.8 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:

```
import DA.Text

multiplyThreeNumbers : Int -> Int -> Int -> Int
multiplyThreeNumbers xx yy zz =
  xx * yy * zz

multiplyTwoNumbersWith7 = multiplyThreeNumbers 7

multiplyWith21 = multiplyTwoNumbersWith7 3
```

You could also define equivalent lambda functions:

```
multiplyWith18 = multiplyThreeNumbers 3 6

multiplyWith18_v2 : Int -> Int
```

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.

```
-- Higher order function
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"]
```

The *Folding* 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:

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.

2.1.2.9 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.

2.1.2.10 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
```

For details on how to upload a Daml archive to the ledger, see the *deploy documentation*. 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.

Importing 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.

Importing 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
```

Importing 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. Furthermore, if the typeclass definition uses the `FunctionalDependencies` language extension, this may cause additional problems, since the functional dependencies cannot be recovered. So this is something to keep in mind when redefining typeclasses and when using `FunctionalDependencies`.
5. Certain advanced type system features cannot be reconstructed. In particular, `DA.Generics` and `DeriveGeneric` cannot be reconstructed. This may result in certain definitions being unavailable when importing a module that uses these advanced features.

Because of their flexibility, data-dependencies are a tool that is recommended for performing Daml model upgrades. See the *upgrade documentation* for more details.

Referencing 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 package 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`.

2.1.2.11 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.

You can use contract keys to stably refer to a contract, even through iterations of instances of it.

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 *Specifying maintainers* below).

It's best to use simple types for your keys like `Text` or `Int`, rather than a list or more complex type.

Specifying 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.

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 contractKey
```

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`). To run `exerciseByKey` you need authorization from the controllers of the choice and at least one stakeholder. This is equivalent to the authorization needed to do a `fetchByKey` followed by an `exercise`.

Example

A complete example of possible success and failure scenarios of *fetchByKey* and *lookupByKey* is shown below.

```
-- Copyright (c) 2022 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
```

(continues on next page)

(continued from previous page)

```
  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]
  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
```

(continues on next page)

(continued from previous page)

```
        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
        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
```

(continues on next page)

(continued from previous page)

```
        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
      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]
```

(continues on next page)

(continued from previous page)

```
-- 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
  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
```

(continues on next page)

(continued from previous page)

```
      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
  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
```

(continues on next page)

(continued from previous page)

```
  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
    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 ()
```

2.1.2.12 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 <> ")"
```

Throwing 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.

Catching 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
      msg = "MyException"
  (ArithmeticError _) ->
    create Error with
      p = p
      msg = "ArithmeticError"
```

2.1.3 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:

2.1.3.1 Module 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**

_interfaceTypeRep : i -> TypeRep

class *HasToInterface* t i **where**

_toInterface : t -> i

class *HasFromInterface* t i **where**

fromInterface : i -> *Optional* t

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 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 *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 *HasExerciseGuarded* t c r **where**

Exposes `exerciseGuarded` function.

exerciseGuarded : (t -> *Bool*) -> *ContractId* t -> c -> *Update* r
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
getTemplateTypeRep	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)

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

instance *Eq Archive*

instance *Show Archive*

type ***Choice*** **t c r** = (*TemplateOrInterface* t, *HasExercise* t c r, *HasToAnyChoice* t c r, *HasFromAnyChoice* t c r)
Constraint satisfied by choices.

type ***Template*** **t** = (*HasSignatory* t, *HasObserver* t, *HasEnsure* t, *HasAgreement* t, *HasCreate* t, *HasFetch* t, *HasArchive* t, *HasTemplateTypeRep* t, *HasToAnyTemplate* t, *HasFromAnyTemplate* t)
Constraint satisfied by templates.

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.

type ***TemplateOrInterface*** **t** = (*HasTemplateTypeRep* t, *HasToAnyTemplate* t, *HasFromAnyTemplate* t)

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*

toInterface : *HasToInterface* t i => t -> i

toInterfaceContractId : *HasToInterface* t i => *ContractId* t -> *ContractId* i

fromInterfaceContractId : (*HasFromInterface* t i, *HasFetch* i) => *ContractId* i -> *Update* (*Optional* (*ContractId* t))

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).

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
`error` stops execution and displays the given error message.
If called within a transaction, it will abort the current transaction. Outside of a transaction (scenarios, Daml Script or Daml Triggers) it will stop the whole scenario/script/trigger.
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`

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.

2.1.3.2 Module 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.

2.1.3.3 Module 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.

2.1.3.4 Module 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)

2.1.3.5 Module 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.

2.1.3.6 Module 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)

2.1.3.7 Module 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
```

2.1.3.8 Module 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.

2.1.3.9 Module 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

2.1.3.10 Module 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*	

2.1.3.11 Module 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* (*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.

2.1.3.12 Module 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 `()`.

2.1.3.13 Module 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.

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.

2.1.3.14 Module 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
```

2.1.3.15 Module 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.

2.1.3.16 Module 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 == Right True` and `toBool false == Right False`. Otherwise, it returns `Left x`, where `x` is the input.

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.

2.1.3.17 Module 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.

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.

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.

2.1.3.18 Module 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

2.1.3.19 Module 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)

2.1.3.20 Module 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.

2.1.3.21 Module 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)

2.1.3.22 Module 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.

2.1.3.23 Module 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.

2.1.3.24 Module 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

2.1.3.25 Module 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)

2.1.3.26 Module 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`.

2.1.3.27 Module 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`.

2.1.3.28 Module 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.

2.1.3.29 Module 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.

2.1.3.30 Module 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*

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

2.1.3.31 Module 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* (*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.

2.1.3.32 Module 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.

2.1.3.33 Module 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 *Applicative* (*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 -> *Text* -> *Validation Text* b
Convert an `Optional t` into a `Validation Text t`, or more generally into an `m t` for any `ActionFail` type `m`.

2.1.4 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`.

Initiate and Accept 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.

Multiple party agreement 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.

Delegation 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.

Authorization The Authorization pattern demonstrates how to make sure a controlling party is authorized before they take certain actions.

Locking 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.

2.1.4.1 Initiate and Accept

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. There is no difference in 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 on-boarding 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 no participants are forced into an obligation.

Implementation

The Initiate and Accept pattern in general involves 2 contracts:

Initiate contract The Initiate contract can be created from a role contract or any other point in the workflow. In this example, initiate contract is the proposal contract *CoinIssueProposal* 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 (e.g. re-negotiate terms).

```
template CoinIssueProposal
  with
    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 = []
```

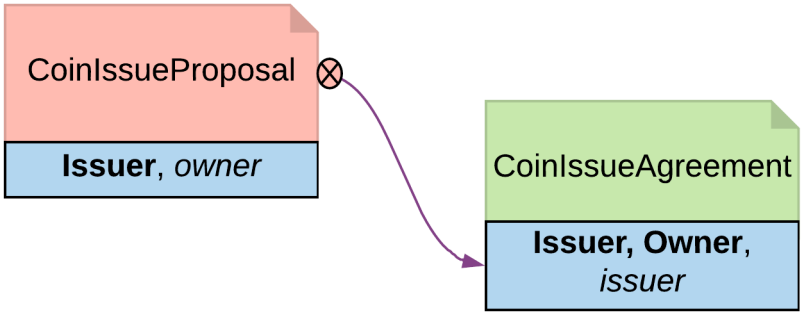


Fig. 1: Initiate and Accept pattern diagram

Trade-offs

Initiate and Accept can be quite verbose if signatures from more than two parties are required to progress the workflow.

2.1.4.2 Multiple party agreement

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 *Initiate and Accept* 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 *Initiate and Accept* 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
```

(continues on next page)

(continued from previous page)

```
      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
```

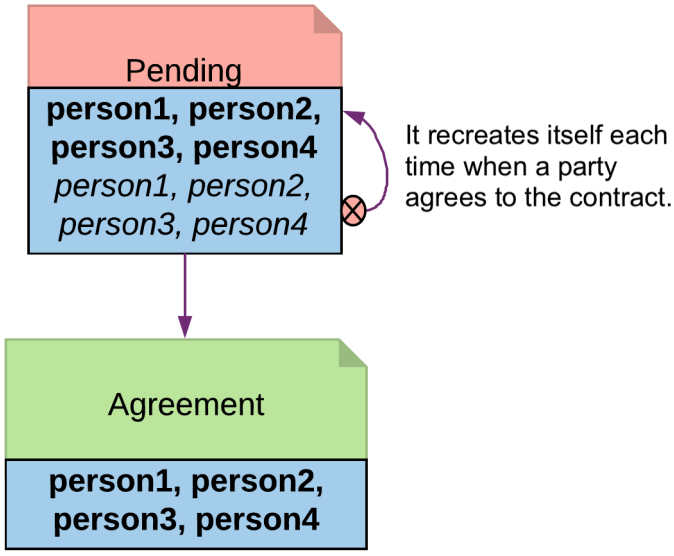


Fig. 2: Multiple Party Agreement Diagram

2.1.4.3 Delegation

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]
```

(continues on next page)

(continued from previous page)

```
  where
    signatory issuer, owner
    observer delegates
```

```
    --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)
```

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

    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.

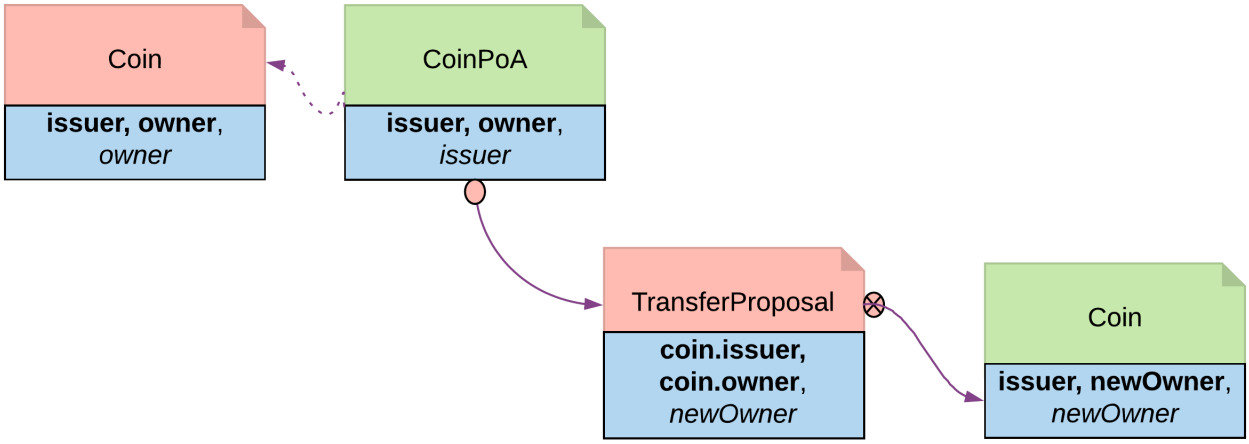


Fig. 3: Delegation pattern diagram

2.1.4.4 Authorization

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
```

```
    --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)
```

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

    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
```

2.1.4.5 Locking

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.

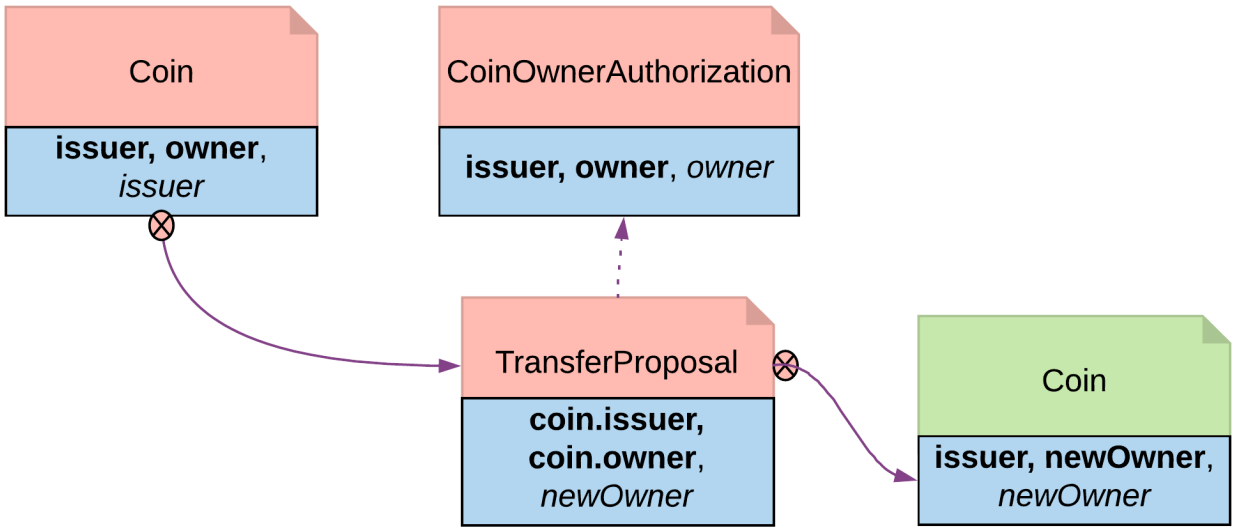


Fig. 4: Authorization Diagram

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:

Locking 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
```

(continues on next page)

(continued from previous page)

```
      controller locker
      do create coin
```

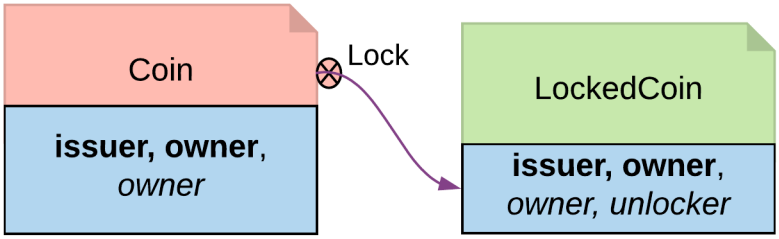


Fig. 5: Locking By Consuming Choice Diagram

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
```

(continues on next page)

(continued from previous page)

```
      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 archivaed
        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
```

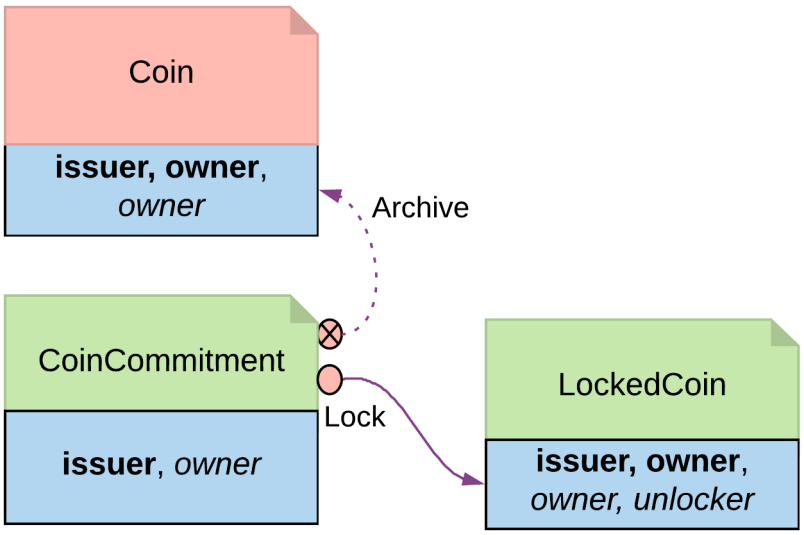


Fig. 6: Locking By Archiving Contract Diagram

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.

Locking 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

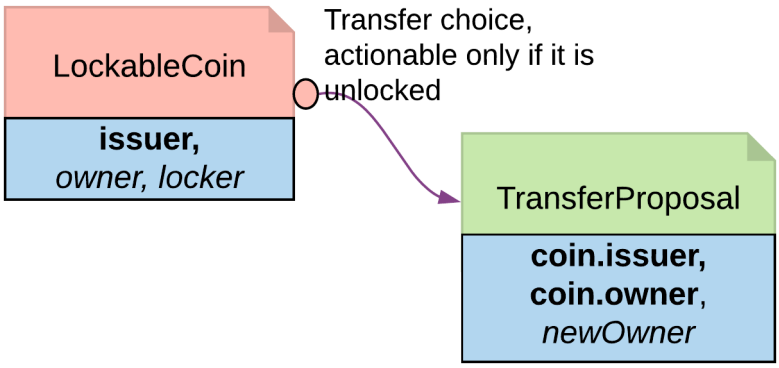
```
template LockableCoin
  with
    owner: Party
    issuer: Party
    amount: Decimal
    locker: Party
```

(continues on next page)

(continued from previous page)

```
  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)
        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.

Locking 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
```

(continues on next page)

(continued from previous page)

```
    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
    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
```

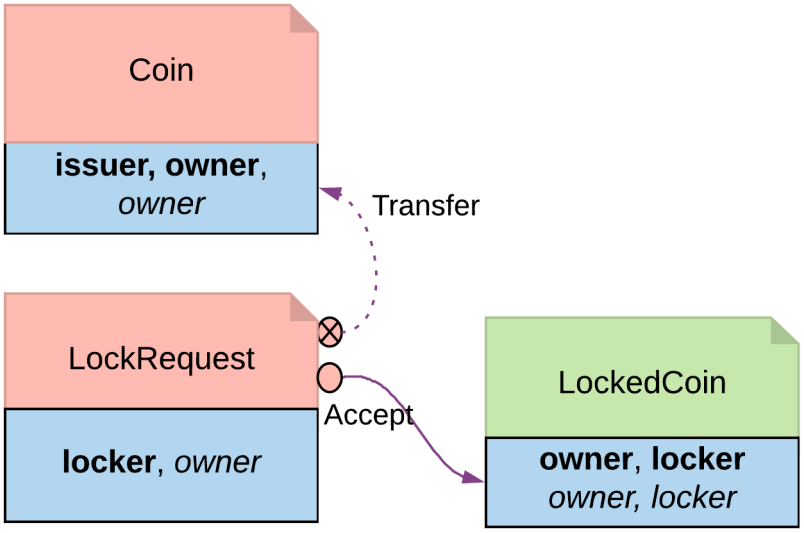


Fig. 7: Locking By Safekeeping Diagram

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.

2.1.4.6 Diagram legends

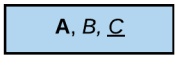
2.2 Building Applications

2.2.1 Application architecture

This section describes our recommended design of a full-stack Daml application.

Active contract

Archived contract



Signatories, *Controllers*, *Observers*

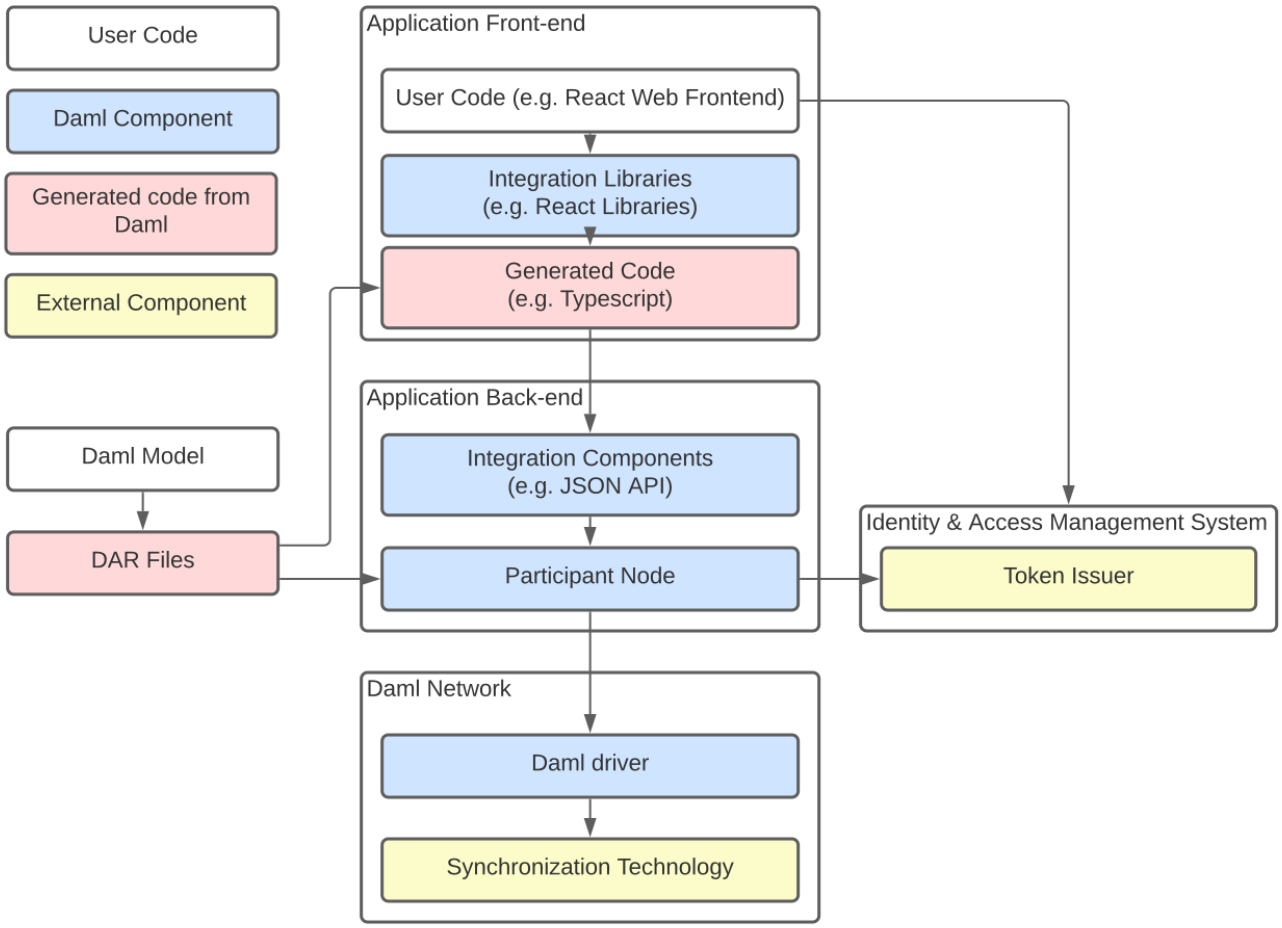
Non-consuming choice

Consuming choice

Consuming choice (but recreating itself with an updated state)

Create another contract from a choice

Reference to contractId



The above image shows the recommended architecture. Of course 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

2.2.1.1 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.

2.2.1.2 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.

2.2.1.3 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.

2.2.1.4 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.

There are several things that 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*.

Dealing 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.

Failing 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.

Dealing 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*.

2.2.2 JavaScript Client Libraries

The JavaScript Client Libraries are the recommended way to build a frontend for a Daml application. The *JavaScript Code Generator* can automatically generate JavaScript containing metadata about Daml packages that is required to use these libraries. We provide an integration for the React framework with the @daml/react library. However, you can choose any JavaScript/TypeScript based framework and use the @daml/ledger library directly to connect and interact with a Daml ledger via its *HTTP JSON API*.

The @daml/types library contains TypeScript data types corresponding to primitive Daml data types, such as `Party` or `Text`. It is used by the @daml/react and @daml/ledger libraries.

2.2.2.1 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.

Usage

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`	`ContractId<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.

2.2.2.2 @daml/react

@daml/react documentation

2.2.2.3 @daml/ledger

@daml/ledger documentation

2.2.2.4 @daml/types

@daml/types documentation

2.2.3 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.

We welcome feedback about the JSON API on our issue tracker, or on our forum.

2.2.3.1 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).

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.

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 the expected Daml-LF type is needed to decide which one.

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"
9999999999999999999999999990
+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`.

2.2.3.2 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

2.2.3.3 Production Setup

The vast majority of the prior documentation focuses on ease of testing and running the service in a dev environment. From a production perspective given the wide variety of use-cases there is far less of an established framework for deploying the *HTTP JSON API* server. In this document we would try to list some recommendations for production deployments.

The *HTTP JSON API* server is a JVM application that by default uses an in-memory backend. This in-memory backend setup is inefficient for larger datasets as for every query it ends up fetching the entire active contract set for the templates referenced in that query. For this reason for production setups at a minimum we recommend to use a database as a query store, this will allow for more efficient caching of the data to improve query performance. Details for enabling a query store are highlighted below.

Query store

Note: Daml Open Source only supports PostgreSQL backends for the *HTTP JSON API* server, but Daml Enterprise also supports Oracle backends.

The query store is a cached search index and is useful for use 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 having to request the entire ACS every time.

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"
}
```

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"
```

Consult your database vendor's JDBC driver documentation to learn how to specify a JDBC connection string that suits your needs.

The `start-mode` is a custom parameter defined by the query store configuration itself which allows to deal with the initialization and usage of the database which backs the query store.

Depending on how you prefer to operate it, you can either choose to:

run the *HTTP JSON API* server with `start-mode=create-only` and a user that has exclusive rights to creating the tables needed for the query store to operate and then start it again with `start-mode=start-only` with a user that can use those tables but not apply schema changes, or
run the *HTTP JSON API* server with a user that can both create and use the query store tables by passing `start-mode=create-and-start`

When restarting the *HTTP JSON API* server after the schema has been already created, it's safe to always use `start-mode=start-only`.

Note: The full list of query store configuration flags supported can be seen by running `daml json-api --help`.

Data continuity

The query store is a cache, which means that it's perfectly fine to drop it as the data it contains it's a subset of what can be safely 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 be able to work with a newer version.

However, the *HTTP JSON API* is able to tolerate working with query stores initialized by a previous version of the software as long as the underlying schema did not change.

The query store keeps track of the schema version under which it was initialized and refuses to start if a new schema is detected when running 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 and create a new one (consult your database vendor's documentation as to how this ought to be done) and then proceed to create and start the server using either `start-mode=create-only` and `start-mode=start-only` or `start-mode=create-and-start` as described above, depending on your preferred production setup.

Security and privacy

For an *HTTP JSON API* server, all data is maintained by the operator of the deployment. Thus, it is their responsibility to ensure that the data abides by the necessary regulations and confidentiality expectations.

It is recommended to use 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.

To protect data in transit and over untrusted networks, the *HTTP JSON API* server provides TLS support, to enable TLS you need to specify the private key for your server and the certificate chain via the below config block specifying the `cert-chain-file`, `private-key-file`, you can also set a custom root CA certificate used to validate client certificates via `trust-collection-file` parameter.

```
ledger-api {
  address = "127.0.0.1"
  port = 6400
```

(continues on next page)

(continued from previous page)

```
  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 refer to this reference implementation

Architecture

Components

A production setup of the *HTTP JSON API* will involve the following components:

the *HTTP JSON API* server
the query store backend database server
the ledger

HTTP JSON API server exposes an API to interact with the Ledger and it uses JDBC to interact with its underlying query store for caching and serving data efficiently.

The *HTTP JSON API* server releases are regularly tested with OpenJDK 11 on a x86_64 architecture, with Ubuntu 20.04, macOS 11.5.2 and Windows Server 2016.

In production, we recommend running on a x86_64 architecture in a Linux environment. This environment should have a Java SE Runtime Environment such as OpenJDK JRE and must be compatible with OpenJDK version 11.0.11 or later. We recommend using PostgreSQL server as query-store, most of our tests have been done with servers running version > 10.

Scaling and Redundancy

Note: This section of the document only talks about scaling and redundancy setup for the *HTTP JSON API* server. In all of the recommendations suggested below we assume that the JSON API always interacts with a single participant on the ledger.

We advise that the *HTTP JSON API* server and query store components to have dedicated computation and memory resources available to them. This can be achieved via containerization or setting them up on independent physical servers. Ensure that the two components are **physically co-located** to

reduce network latency for communication. The scaling and availability aspects heavily rely on the interactions between the core components listed above.

With respect to scaling we recommend to follow the general advice in trying to understand the bottlenecks and see if adding additional processing power/memory is beneficial.

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 app orchestration utilities which monitor the health of the service and perform subsequent operations to ensure availability. These utilities can use the healthcheck endpoints provided by the *HTTP JSON API* server. This can also be tied into supporting arbitrary autoscaling implementation to ensure minimum number of *HTTP JSON API* servers on failures.

Logging

HTTP JSON API server uses the industry-standard Logback for logging. You can read more about that 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 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 to have rich logs 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]".
```

(continues on next page)

(continued from previous page)

```
  reporter = "console"
  //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`: metrics are pre-aggregated on the *HTTP JSON API* and sent to the reporter, this option allows the user to set the interval. 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 all metrics registered by a meter and by a histogram, where the histogram records the time necessary to execute a given operation (in fractional milliseconds).

List of metrics

The following is a list of selected metrics that can be particularly important to track.

`daml.http_json_api.command_submission_timing`

A timer. Measures latency (in milliseconds) for processing of a command submission request.

`daml.http_json_api.query_all_timing`

A timer. Measures latency (in milliseconds) for processing of a query GET request.

`daml.http_json_api.query_matching_timing`

A timer. Measures latency (in milliseconds) for processing of a query POST request.

`daml.http_json_api.fetch_timing`

A timer. Measures latency (in milliseconds) for processing of a fetch request.

`daml.http_json_api.get_party_timing`

A timer. Measures latency (in milliseconds) for processing of a get party/parties request.

`daml.http_json_api.allocate_party_timing`

A timer. Measures latency (in milliseconds) for processing of a party management request.

`daml.http_json_api.download_package_timing`

A timer. Measures latency (in milliseconds) for processing of a package download request.

`daml.http_json_api.upload_package_timing`

A timer. Measures latency (in milliseconds) for processing of a package upload request.

`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.http_request_throughput`

A meter. Number of http requests

`daml.http_json_api.websocket_request_count`

A Counter. Count of active websocket connections

`daml.http_json_api.command_submission_throughput`

A meter. Number of command submissions

`daml.http_json_api.upload_packages_throughput`

A meter. Number of package uploads

`daml.http_json_api.allocation_party_throughput`

A meter. Number of party allocations

2.2.3.4 Running 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 --ledgerid MyLedger --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 {
    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.
```

(continues on next page)

(continued from previous page)

```
  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
  }

  // 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 JSON API to use a PostgreSQL backend as a cache. 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 PostgreSQL backend acts purely as a cache. It is safe to reinitialize the database at any time.

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 = "create-if-needed-and-start"
}
```

Note: When you use the Query Store you'll want to use `start-mode=create-if-needed-and-start` so that all the necessary tables are created if they don't exist.

you can also use the `--query-store-jdbc-config` CLI flag (deprecated), an example of which is 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=create-
↪if-needed-and-start"
```

Note: The JSON API provides many other useful configuration flags, run `daml json-api --help` to see all of them.

Access Tokens

The JSON API essentially performs two separate tasks:

1. It talks to the Ledger API to get data it needs to operate, for this you need to *provide an access token* if your Ledger requires authorization. Learn more in the *Authorization* docs.
2. It accepts requests from Parties and passes them on to the Ledger API, for this each party needs to provide an *access token with each request* it sends to the JSON API.

Note: By default, the Daml Sandbox does not does not require access tokens. However, you still need to provide a party-specific access token when submitting commands or queries as a party. The token will not be validated in this case but it will be decoded to extract information like the party submitting the command.

Party-specific Access Tokens

Party-specific requests, i.e., command submissions and queries, require a JWT with some additional restrictions compared to general *Daml ledger access token formats*. For command submissions, `actAs` must contain at least one party and `readAs` can contain 0 or more parties. Queries require at least one party in either `actAs` or `readAs`. The application id and ledger id are mandatory. HTTP requests pass the token in a header, while WebSocket requests pass the token in a subprotocol.

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 the JWT format documented below.

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": {
    "ledgerId": "MyLedger",
    "applicationId": "foobar",
    "actAs": ["Alice"]
```

(continues on next page)

(continued from previous page)

```
  }
}
```

The value of the `ledgerId` field has to match the `ledgerId` of your underlying Daml Ledger. For the Sandbox this corresponds to the `--ledgerid MyLedger` flag.

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 the example which uses `Alice` for a party. Each request can only be for one party. For example you couldn't have `actAs` defined as `["Alice", "Bob"]`.

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": "MyLedger", "applicationId": "HTTP-JSON-API-Gateway", "actAs": ["Alice"]}}`:


```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJodHRwczovL2RhbWwuY29tL2xlZGdlci1hcGkiOnsibGVkZ2VySWQiOiJNeUxlZGdlciIsImFwcGxpY2F0aW9uS
↪34zzF_fbWv7p60r5s1kKzwndvGdsJDX-W4Xhm4oVdpk
```


`{"https://daml.com/ledger-api": {"ledgerId": "MyLedger", "applicationId": "HTTP-JSON-API-Gateway", "actAs": ["Bob"]}}`:


```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJodHRwczovL2RhbWwuY29tL2xlZGdlci1hcGkiOnsibGVkZ2VySWQiOiJNeUxlZGdlciIsImFwcGxpY2F0aW9uS
↪0uPPZtM1AmKvnGixt_Qo53cMDcpnziCjKKiWLvMX2VM
```


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
```

2.2.3.5 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, either an `errors` or `result` field and an optional `warnings`:

```
{
    "status": <400 | 401 | 403 | 404 | 409 | 500 | 503 | 504>,
    "errors": <JSON array of strings>, | "result": <JSON object or array>,
    ["warnings": <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.

See the following blog post for more details about error handling best practices: REST API Error Codes 101.

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>
}
```

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"]}
```

2.2.3.6 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": "Iou:Iou",
  "payload": {
    "issuer": "Alice",
    "owner": "Alice",
    "currency": "USD",
    "amount": "999.99",
    "observers": []
  }
}
```

Where:

`templateId` is the contract template identifier, which can be formatted as either:
- "`<package ID>:<module>:<entity>`" or
- "`<module>:<entity>`" if contract template can be uniquely identified by its module and entity name.

`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"
        ],
        "contractId": "#124:0",
        "templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:Iou"
    }
}
```

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).

2.2.3.7 Creating a Contract with a Command ID

When creating a new contract you may specify an optional `meta` field. This allows you to control the `commandId`, `actAs`, and `readAs` 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": "Iou:Iou",
```

(continues on next page)

(continued from previous page)

```
  "payload": {
    "observers": [],
    "issuer": "Alice",
    "amount": "999.99",
    "currency": "USD",
    "owner": "Alice"
  },
  "meta": {
    "commandId": "a unique ID",
    "actAs": ["Alice"],
    "readAs": ["PublicParty"]
  }
}
```

Where:

`commandId` – optional field, a unique string identifying the command.

2.2.3.8 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": "Iou:Iou",
    "contractId": "#124:0",
    "choice": "Iou_Transfer",
    "argument": {
        "newOwner": "Alice"
    }
}
```

Where:

`templateId` – contract template identifier, same as in *create request*,
`contractId` – contract identifier, the value from the *create response*,
`choice` – Daml contract choice, that is being exercised,
`argument` – contract choice argument(s).

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
↪"
                }
            }
        ]
    }
}
```

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.

2.2.3.9 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": "Account:Account",
    "key": {
        "_1": "Alice",
        "_2": "abc123"
    },
    "choice": "Archive",
    "argument": {}
}
```

Where:

`templateId` – contract template identifier, same as in *create request*,
`key` – contract key, formatted according to the *Daml-LF JSON Encoding*,
`choice` – Daml contract choice, that is being exercised,
`argument` – contract choice argument(s), empty, because `Archive` does not take any.

HTTP Response

Formatted similar to *Exercise by Contract ID response*.

2.2.3.10 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": "Iou:Iou",
  "payload": {
    "observers": [],
    "issuer": "Alice",
    "amount": "999.99",
    "currency": "USD",
    "owner": "Alice"
  },
  "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*,
`choice` – Daml contract choice, that is being exercised,
`argument` – contract choice argument(s).

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": "",
          "payload": {
            "iou": {
              "observers": [],
              "issuer": "Alice",
              "amount": "999.99",
              "currency": "USD",
              "owner": "Alice"
            },
            "newOwner": "Bob"
          },
          "signatories": [
            "Alice"
          ],
          "contractId": "#1:2",
          "templateId":
↪"a3b788b4dc18dc060bfb82366ae6dc055b1e361d646d5cfdb1b729607e344336:Iou:IouTransfer
↪"
        }
      }
    ]
```


(continues on next page)

(continued from previous page)

```
  },
  "status": 200
}
```

2.2.3.11 Fetch Contract by Contract ID

HTTP Request

URL: `/v1/fetch`
Method: `POST`
Content-Type: `application/json`
Content:

application/json body:

```
{
  "contractId": "#201:1"
}
```

`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": {
            "iou": {
                "observers": [],
                "issuer": "Alice",
                "amount": "999.99",
                "currency": "USD",
                "owner": "Alice"
            },
```

(continues on next page)

(continued from previous page)

```
            "newOwner": "Alice"
        },
        "signatories": [
            "Alice"
        ],
        "contractId": "#201:1",
        "templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:IouTransfer
↪"
    }
}
```

2.2.3.12 Fetch Contract by Key

HTTP Request

URL: `/v1/fetch`
Method: `POST`
Content-Type: `application/json`
Content:

```
{
    "templateId": "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",
            "_2": "abc123"
        },
        "contractId": "#697:0",
        "templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Account:Account
↪"
    }
}
```

2.2.3.13 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.

2.2.3.14 Get all Active Contracts Matching a Given Query

List currently active contracts that match a given query.

HTTP Request

URL: `/v1/query`
Method: `POST`
Content-Type: `application/json`
Content:

```
{
    "templateIds": ["Iou:Iou"],
    "query": {"amount": 999.99},
    "readers": ["Alice"]
}
```

Where:

`templateIds` – an array of contract template identifiers to search through,
`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": [
                "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": [
```

(continues on next page)

(continued from previous page)

```
                "Alice"
            ],
            "contractId": "#52:0",
            "templateId":
↪"b10d22d6c2f2fae41b353315cf893ed66996ecb0abe4424ea6a81576918f658a:Iou:Iou"
        }
    ],
    "status": 200
}
```

2.2.3.15 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
    }
```

(continues on next page)

(continued from previous page)

```
  ]
}
```

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.

2.2.3.16 Fetch All Known Parties

URL: `/v1/parties`
Method: `GET`
Content: <EMPTY>

HTTP Response

The response is the same as for the POST method above.

2.2.3.17 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
}
```

2.2.3.18 Creating 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
}
```

2.2.3.19 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
}
```

2.2.3.20 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
}
```

2.2.3.21 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
}
```

2.2.3.22 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
}
```

2.2.3.23 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"
    }
  ]
}
```

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"
    }
```

(continues on next page)

(continued from previous page)

```
  ],
  "status": 200
}
```

Returns the rights that were newly granted.

2.2.3.24 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",
      "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",
```

(continues on next page)

(continued from previous page)

```
      "party": "Alice",
    },
    {
      "type": "CanReadAs",
      "party": "Bob",
    },
    {
      "type": "ParticipantAdmin"
    }
  ],
  "status": 200
}
```

Returns the rights that were actually granted.

2.2.3.25 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
}
```

2.2.3.26 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
}
```

2.2.3.27 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.

2.2.3.28 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
}
```

2.2.3.29 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
}
```

2.2.3.30 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() {
```

(continues on next page)

(continued from previous page)

```
  ws.send(JSON.stringify({templateIds: ["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.

`application/json` body must be sent first, formatted according to the *Query language*:

```
{"templateIds": ["Iou:Iou"]}
```

Multiple queries may be specified in an array, for overlapping or different sets of template IDs:

```
[
    {"templateIds": ["Iou:Iou"], "query": {"amount": {"%lte": 50}}},
    {"templateIds": ["Iou:Iou"], "query": {"amount": {"%gt": 50}}},
    {"templateIds": ["Iou:Iou"]}
]
```

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": ["Iou:Iou"], "query": {"amount": {"%lte": 50}}},
    {"templateIds": ["Iou:Iou"], "query": {"amount": {"%gt": 50}}},
    {"templateIds": ["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.

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",
```

(continues on next page)

(continued from previous page)

```
                "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": [{
```

(continues on next page)

(continued from previous page)

```
        "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.

`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": "Account:Account", "key": {"_1": "Alice", "_2": "abc123"}},
    {"templateId": "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": "Account:Account", "key": {"_1": "Alice", "_2": "abc123"},
     "contractIdAtOffset": "#1:0"},
    {"templateId": "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.

2.2.3.31 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 liveness probe. It validates both the ledger connection as well as the database connection.

2.2.4 Daml Script

2.2.4.1 Daml Script Library

The Daml Script library defines the API used to implement Daml scripts. See *Daml Script*:: for more information on Daml script.

Module 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, 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, 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, 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. This is semantically equivalent to calling `query` and filtering on the client side.

queryContractKey : (HasCallStack, TemplateKey t k, IsParties p) => p -> k -> *Script* (Optional (ContractId t, t))

setTime : HasCallStack => Time -> *Script* ()
Set the time via the time service.
This is only supported in static time mode when running over the gRPC API and in Daml Studio. 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.
Only supported in static time mode when running over the gRPC API and in Daml Studio. 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 wallcock 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 => 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) => 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.

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 load all the example code by running `daml new script-example --template script-example`

2.2.4.2 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 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 return something 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 functions 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)
```

Running 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 ScriptExample: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
↪",
  "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. It takes a `LedgerParties` argument, so a valid file for `--input-file` would look like:

Using the previosuly 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 Scrip-
tExample:initialize --ledger-host localhost --ledger-port 6865 --input-file
ledger-parties.json
```

2.2.4.3 Using Daml Script for Ledger Initialization

You can use Daml script to initialize a ledger on startup. To do so, specify an `init-script: ScriptExample: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
```

Migrating 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`.

2.2.4.4 Using Daml Script in Distributed Topologies

So far, we have run Daml script against a single participant node. It is also more possible to run it in a setting where different parties are hosted on different participant nodes. To do so, pass the `--participant-config participants.json` file to `daml script` instead of `--ledger-host` and `ledger-port`. 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": 6865, "access_token": "jwt_for_alice
↪", "application_id": "myapp"},
        "two": {"host": "localhost", "port": 6865, "access_token": "jwt_for_bob",
↪"application_id": "myapp"}
```

(continues on next page)

(continued from previous page)

```
    },
    "party_participants": {"alice": "one", "bob": "two"}
}
```

This will define a participant called `one`, a default participant and it defines that the party `alice` is 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 `default_participant` is not specified, 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.

2.2.4.5 Running 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 use the `access_token` field in the participant config specified via `--participant-config`. The section on *using Daml Script in distributed topologies* contains an example. Note that you can specify the same participant twice if you want different auth tokens.

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.

2.2.4.6 Running 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 usecase, 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.

2.2.5 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.

2.2.5.1 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
[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`.

2.2.5.2 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)
```

2.2.5.3 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
```

2.2.5.4 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.

2.2.5.5 Connection to a Ledger with Authorization

If your ledger requires an authorization token you can pass it via `--access-token-file`.

2.2.5.6 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"}]
```

2.2.6 Upgrading and Extending Daml applications

2.2.6.1 Extending Daml applications

Note: Cross-SDK extensions require Daml-LF 1.8 or newer. This is the default starting from SDK 1.0. For older releases add `build-options: ["--target=1.8"]` to your `daml.yaml` to select Daml-LF 1.8.

Consider the following simple Daml model for carbon certificates:

```
module CarbonV1 where

template CarbonCert
  with
```

(continues on next page)

(continued from previous page)

```
    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.

2.2.6.2 Upgrading Daml applications

Note: Cross-SDK upgrades require Daml-LF 1.8 or newer. This is the default starting from SDK 1.0. For older releases add `build-options: ["--target=1.8"]` to your `daml.yaml` to select Daml-LF 1.8.

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.

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.

Structuring 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
         create CarbonCertWithMethod with
           issuer = cert.issuer
           owner = cert.owner
           carbon_metric_tons = cert.carbon_metric_tons
           carbon_offset_method = "unknown"
```

Building and deploying 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
```

Create some 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
↪",
  "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.

Building and deploying 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 crypto-

graphically 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
```

Building and deploying 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
```

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.

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*.

2.2.6.3 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.

Structuring 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`.

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)
```

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) <$>
```

(continues on next page)

(continued from previous page)

```
    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.

Deploying and Executing 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.

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.

2.2.7 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*. They check for every Ledger API request whether the request contains an access token that is valid and sufficient to authorize the request. You thus need to add support for authorization using access token to your application to run it against a deployed ledger.

2.2.7.1 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*.

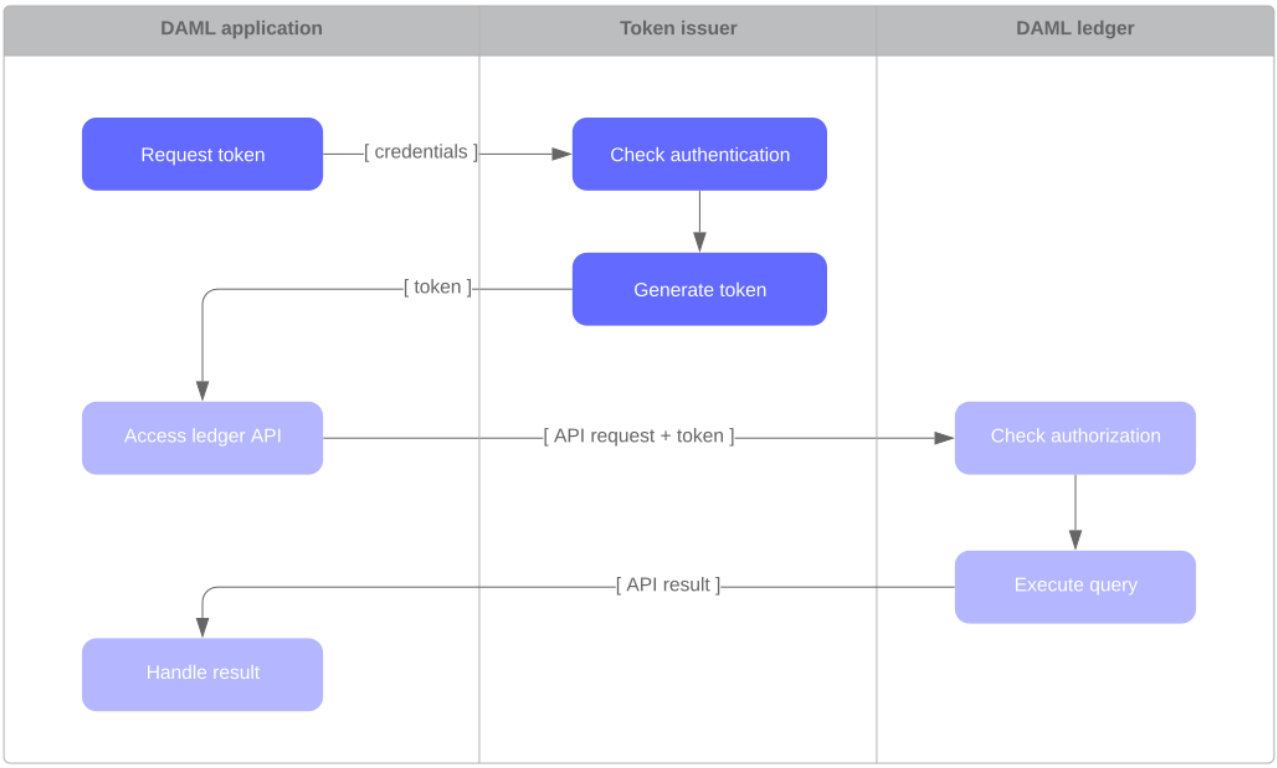
Whether a participant node *can* serve such a request depends on whether the participant node hosts the respective parties, and whether the request is valid according to the *Daml Ledger Model*. Whether a participant node *will* serve such a request to a Daml application depends on whether the request includes an access token that is valid and sufficient to authorize the request for this participant node.

2.2.7.2 Acquiring and using access tokens

How an application should acquire access tokens depends on the participant node it talks to and is ultimately setup by the participant node operator. Many setups use a flow in the style of OAuth 2.0:

First, the Daml application 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.

Then, the Daml application sends the access token along with every Ledger API request. The Daml ledger verifies the signature of the token to make sure it has not been tampered with and was issued by one of its trusted token issuers, and then checks that the token has not yet expired and that the privileges described in the token authorize the given Ledger API request.



As shown above, using access tokens requires your application to attach them to every request. How to do that 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. Here is for example the relevant documentation for the *Java bindings* and the *JSON API*.

2.2.7.3 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 adminstrate the participant node
> `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)
CommandCompletion-Service	CompletionEnd	public
	CompletionStream	for each requested party p: canReadAs(p)
CommandSubmission-Service	Submit	for submitting party p: canActAs(p)
CommandService	All	for submitting party p: canActAs(p)
Health	All	no access token required for health checking
LedgerConfigurationSer-vice	GetLedgerConfigura-tion	public
MeteringReportService	All	participant_admin
PackageService	All	public
PackageManagementSer-vice	All	participant_admin
PartyManagementService	All	participant_admin
ParticipantPruningSer-vice	All	participant_admin
ServerReflection	All	no access token required for gRPC service re-flection
TimeService	GetTime	public
	SetTime	participant_admin
TransactionService	LedgerEnd	public
	All (except LedgerEnd)	for each requested party p: canReadAs(p)
UserManagementService	All	participant_admin
	GetUser	authenticated users can get their own user
	ListUserRights	authenticated users can list their own rights
VersionService	All	public

2.2.7.4 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 in use 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 explained below.

User access tokens are JWTs that follow the OAuth 2.0 standard with a JSON payload of the following format.

```
{
   "aud": "someParticipantId",
   "sub": "someUserId",
   "exp": 1300819380
   "scope": "daml_ledger_api"
}
```

The above notations are explained below:

`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
`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

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 for the custom claims to be present at the same level as `exp` in the token above, instead of being nested below `"https://daml.com/ledger-api"`.

2.2.8 The Ledger API

2.2.8.1 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`.

Submitting 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*.

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*.

Reading 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`.

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.

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.

Utility services

Party management service

Use the **party management service** to allocate parties on the ledger 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.

In contrast to parties, users are local to a participant node. 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; and 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, and delete users and their rights. See the *UserManagementFeature descriptor* to learn about 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*.

Read the *Authorization documentation* to understand how Ledger API requests are authorized, and how to use user management to dynamically change an application's rights.

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*.

User Management service

Use the **user management service** to manage users and their rights on a given participant.

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*.

2.2.8.2 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.

Getting 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*

2.2.8.3 Error Codes

Overview

The majority of the errors are a result of some request processing. They are logged and returned to the user as a failed gRPC response containing the status code, an optional status message and optional metadata.

This approach remains unchanged in principle while we aim at enhancing it 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.

The goal is to enable users, developers and operators to act on the encountered errors in a self-service manner, either in an automated-way or manually.

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. Map to exactly one gRPC status code. We recommended to deal with errors based on their error category. However, if error category itself is too generic you can act on particular error codes.

Correlation id A value whose purpose is to allow the user to clearly identify the request, such that the operator can lookup any log information associated with this error. We use request's submission id for correlation id.

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.

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-cased 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 an 8 character long prefix 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 character that serves as a separator for the machine and human readable parts.
`<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.
```

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 in seconds.
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..) will contain these. Particular resources are implementation specific and vary across ledger implementations.

Many errors will include more information, but there is no guarantee given that additional information will be preserved across versions.

Working 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 error code id.
        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
```

(continues on next page)

(continued from previous page)

```
        // 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])
              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)
          }.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
        }
    }
  }
}
```

Error Categories Inventory

The error categories allow 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.

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.

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.

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.

SystemInternalAssumptionViolated

Category id: 4

gRPC status code: INTERNAL

Default log level: ERROR

Description: Request processing failed due to a violation of system internal invariants.

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.

MaliciousOrFaultyBehaviour

Category id: 5

gRPC status code: UNKNOWN

Default log level: WARN

Description: Request processing failed due to unrecoverable data loss or corruption (e.g. detected via checksums)

Resolution: Expectation: this can be a severe issue that requires operator attention or intervention, and potentially vendor support.

Retry strategy: Retry after operator intervention.

AuthInterceptorInvalidAuthenticationCredentials

Category id: 6

gRPC status code: UNAUTHENTICATED

Default log level: WARN

Description: The request does not have valid authentication credentials for the operation.

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.

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.

Resolution: Expectation: this is an application bug or application misconfiguration. Resolution requires application operator intervention.

Retry strategy: Retry after application operator intervention.

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.

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.

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).

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).

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.

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.

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.

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.

Error Codes Inventory

1. KVErrors

Errors that are specific to ledgers based on the KV architecture: Daml Sandbox and VMBC.

1.1. KVErrors / Consistency

Errors that highlight transaction consistency issues in the committer context.

VALIDATION_FAILURE

Explanation: Validation of a transaction submission failed using on-ledger data.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message

Resolution: Either some input contracts have been pruned or the participant is misbehaving.

1.2. KVErrors / Internal

Errors that arise from an internal system misbehavior.

INVALID_PARTICIPANT_STATE

Explanation: An invalid participant state has been detected.

Category: SystemInternalAssumptionViolated

Conveyance: This error is logged with log-level ERROR on the server side. This error is exposed on the API with grpc-status INTERNAL without any details due to security reasons

Resolution: Contact support.

MISSING_INPUT_STATE

Explanation: The participant didn't provide a necessary transaction submission input.

Category: SystemInternalAssumptionViolated

Conveyance: This error is logged with log-level ERROR on the server side. This error is exposed on the API with grpc-status INTERNAL without any details due to security reasons

Resolution: Contact support.

REJECTION_REASON_NOT_SET

Explanation: A rejection reason has not been set.

Category: SystemInternalAssumptionViolated

Conveyance: This error is logged with log-level ERROR on the server side. This error is exposed on the API with grpc-status INTERNAL without any details due to security reasons

Resolution: Contact support.

SUBMISSION_FAILED

Explanation: An unexpected error occurred while submitting a command to the ledger.

Category: SystemInternalAssumptionViolated

Conveyance: This error is logged with log-level ERROR on the server side. This error is exposed on the API with grpc-status INTERNAL without any details due to security reasons

Resolution: Contact support.

1.3. KVErrors / Resources

Errors that relate to system resources.

RESOURCE_EXHAUSTED

Explanation: A system resource has been exhausted.

Category: ContentionOnSharedResources

Conveyance: This error is logged with log-level INFO on the server side. This error is exposed on the API with grpc-status ABORTED including a detailed error message

Resolution: Retry the transaction submission or provide the details to the participant operator.

1.4. KVErrors / Time

Errors that relate to the Daml concepts of time.

CAUSAL_MONOTONICITY_VIOLATED

Explanation: At least one input contract's ledger time is later than that of the submitted transaction.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message

Resolution: Retry the transaction submission.

INVALID_RECORD_TIME

Explanation: The record time is not within bounds for reasons other than deduplication, such as excessive latency. Excessive clock skew between the participant and the committer or a time model that is too restrictive may also produce this rejection.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message

Resolution: Retry the submission or contact the participant operator.

RECORD_TIME_OUT_OF_RANGE

Explanation: The record time is not within bounds for reasons other than deduplication, such as excessive latency. Excessive clock skew between the participant and the committer or a time model that is too restrictive may also produce this rejection.

Category: InvalidGivenCurrentSystemStateOther

Conveyance: This error is logged with log-level INFO on the server side. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message

Resolution: Retry the transaction submission or contact the participant operator.

2. ParticipantErrorGroup

2.1. ParticipantErrorGroup / IndexErrors

Errors raised by the Participant Index persistence layer.

2.1.1. 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. This error is exposed on the API with grpc-status INTERNAL without any details due to 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. This error is exposed on the API with grpc-status INTERNAL without any details due to 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. This error is exposed on the API with grpc-status UNAVAILABLE including a detailed error message

Resolution: Re-submit the request.

2.2. ParticipantErrorGroup / LedgerApiErrors

Errors raised by or forwarded by the Ledger API.

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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message

Resolution: Fetch newest configuration and/or retry.

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. This error is exposed on the API with grpc-status INTERNAL without any details due to 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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message

Resolution: Refer to the detailed message of the received error.

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 WARN on the server side. This error is 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.

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. This error is 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.

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. This error is 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.

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. This error is 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.

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. This error is exposed on the API with grpc-status UNIMPLEMENTED without any details due to security reasons

Resolution: This error is caused by a participant node misconfiguration or by an implementation bug. Resolution requires participant operator intervention.

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. This error is 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. This error is 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.

2.2.1. 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. This error is exposed on the API with grpc-status INTERNAL without any details due to 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. This error is exposed on the API with grpc-status PERMISSION_DENIED without any details due to 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. This error is 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. This error is exposed on the API with grpc-status UNAUTHENTICATED without any details due to security reasons

Resolution: Ask your participant operator to provide you with an appropriate JWT token.

2.2.2. 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. This error is exposed on the API with grpc-status ABORTED including a detailed error message

Resolution: Retry the transaction submission.

2.2.2.1. ParticipantErrorGroup / LedgerApiErrors / CommandExecution / Interpreter

Errors raised during the command interpretation phase of the command submission evaluation.

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. This error is exposed on the API with grpc-status NOT_FOUND including a detailed error message

Resolution: This error indicates an application error.

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. This error is 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. This error is 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. This error is 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.

2.2.2.1.1. 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. This error is 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.

2.2.2.2. 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. This error is 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: MaliciousOrFaultyBehaviour

Conveyance: This error is logged with log-level WARN on the server side. This error is exposed on the API with grpc-status UNKNOWN without any details due to security reasons

Resolution: Contact support.

2.2.2.3. 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. This error is exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message

Resolution: Inspect error details and correct your application.

2.2.3. 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. This error is 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.

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. This error is 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. This error is 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. This error is 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. This error is 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. This error is 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. This error is 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. This error is 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.

2.2.4. 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. This error is 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. This error is 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. This error is exposed on the API with grpc-status INTERNAL without any details due to security reasons

Resolution: Inspect the error message and contact support.

2.2.4.1. 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. This error is 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. This error is 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. This error is 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. This error is 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. This error is 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. This error is exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message

Resolution: Inspect the dar and contact support.

2.2.5. 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. This error is 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. This error is 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. This error is 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. This error is 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. This error is 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. This error is 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. This error is 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. This error is 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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message

Resolution: Use an offset that is after the pruning offset.

2.2.5.1. 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. This error is 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. This error is 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.

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. This error is 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.

2.2.6. 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.

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. This error is exposed on the API with grpc-status INTERNAL without any details due to security reasons

Resolution: Contact support.

OUT_OF_QUOTA

Deprecation: Corresponds to transaction submission rejections that are not produced anymore.

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. This error is 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. This error is 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. This error is exposed on the API with grpc-status PERMISSION_DENIED without any details due to 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. This error is 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.

2.2.6.1. 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. This error is exposed on the API with grpc-status INTERNAL without any details due to 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. This error is exposed on the API with grpc-status INTERNAL without any details due to security reasons

Resolution: Contact support.

2.2.8.4 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

GetActiveContractsResponse

Field	Type	Label	Description
offset	*string*		Included 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`. Required
workflow_id	*string*		The workflow that created the contracts. Must be a valid LedgerString (as described in `value.proto`). Optional
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`). Optional

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.

Method name	Request type	Response type	Description
GetActiveContracts	*GetActiveContractsRequest*	*GetActiveContractsResponse*	Returns a stream of the latest snapshot of active contracts. If there are no active contracts, the stream returns a single GetActiveContractsResponse 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. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `NOT_FOUND`: if the request does not include a valid ledger id - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields (filters by party cannot be empty)

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.

Method name	Request type	Response type	Description
GetTimeModel	*GetTimeModelRequest*	*GetTimeModelResponse*	Return the currently active time model and the current configuration generation. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation
SetTimeModel	*SetTimeModelRequest*	*SetTimeModelResponse*	Set the ledger time model. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `INVALID_ARGUMENT`: if arguments are invalid, or the provided configuration generation does not match the current active configuration generation. The caller is expected to retry by again fetching current time model using 'GetTimeModel', applying changes and resubmitting. - `DEADLINE_EXCEEDED`: if the request times out. Note that a timed out request may have still been committed to the ledger. Application should re-query the current time model before retrying. - `FAILED_PRECONDITION`: if the request is rejected. - `UNIMPLEMENTED`: if this method is not supported by the backing ledger.

com/daml/ledger/api/v1/admin/metering_report_service.proto

ApplicationMeteringReport

Field	Type	Label	Description
application_id	*string*		The application Id
event_count	*int64*		The event count for the application; i.e., the number of fetch, lookup-by-key, create, and exercise events in transactions issued by this application.

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.
participant_report	*ParticipantMeteringReport*		The computed report.
report_generation_time	google.protobuf.Timestamp		The time at which the report was computed.

ParticipantMeteringReport

Field	Type	Label	Description
participant_id	*string*		The reporting participant
is_final	*bool*		If the report is final it has been generated based on aggregated data so will never change in the future.
application_reports	*ApplicationMeteringReport*	repeated	Per application reports.

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/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. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation
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 Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `UNIMPLEMENTED`: if DAR package uploading is not supported by the backing participant - `INVALID_ARGUMENT`: if the DAR file is too big or malformed. The maximum supported size is implementation specific.

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 one of the following errors: - `INVALID_ARGUMENT`: if the payload, particularly the offset is malformed or missing - `UNIMPLEMENTED`: if the participant is based on a ledger that has not implemented pruning - `INTERNAL`: if the participant has encountered a failure and has potentially applied pruning partially. Such cases warrant verifying the participant health before retrying the prune with the same (or a larger, valid) offset. Successful retries after such errors ensure that different components reach a consistent pruning state. - `FAILED_PRECONDITION`: if the participant is not yet able to prune at the specified offset.

com/daml/ledger/api/v1/admin/party_management_service.proto

AllocatePartyRequest

Field	Type	Label	Description
party_id_hint	*string*		A hint to the backing 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. Optional

AllocatePartyResponse

Field	Type	Label	Description
party_details	*PartyDetails*		

GetParticipantIdRequest

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

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

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

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. Caution, it might not be unique. Optional
is_local	*bool*		true if party is hosted by the backing participant. Required

PartyManagementService

Status: experimental interface, will change before it is deemed production ready

Inspect the party management state of a ledger participant and modify the parts that are modifiable. We use 'backing participant' to refer to this specific participant in the methods of this API.

Method name	Request type	Response type	Description
GetParticipantId	*GetParticipantIdRequest*	*GetParticipantIdResponse*	Return the identifier of the backing 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 backing participant Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation
GetParties	*GetPartiesRequest*	*GetPartiesResponse*	Get the party details of the given parties. Only known parties will be returned in the list. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation
ListKnownParties	*ListKnownPartiesRequest*	*ListKnownPartiesResponse*	List the parties known by the backing participant. The list returned contains parties whose ledger access is facilitated by backing participant and the ones maintained elsewhere. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation
AllocateParty	*AllocatePartyRequest*	*AllocatePartyResponse*	Adds a new party to the set managed by the backing participant. Caller specifies a party identifier suggestion, the actual identifier allocated might be different and is implementation specific. This call may: - Succeed, in which case the actual allocated identifier is visible in the response. - Respond with a gRPC error Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `UNIMPLEMENTED`: if synchronous party allocation is not supported by the backing participant - `DEADLINE_EXCEEDED`: if the request times out - `INVALID_ARGUMENT`: if the provided hint and/or display name is invalid on the given ledger (see below). 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

com/daml/ledger/api/v1/admin/user_management_service.proto

CreateUserRequest

Required authorization: `HasRight(ParticipantAdmin)`

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`. Required

CreateUserResponse

Field	Type	Label	Description
user	*User*		Created user.

DeleteUserRequest

Required authorization: `HasRight(ParticipantAdmin)`

Field	Type	Label	Description
user_id	*string*		The user to delete. Required

DeleteUserResponse

Does not (yet) contain any data.

GetUserRequest

Required authorization: `HasRight(ParticipantAdmin)` 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. Required

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)`

Field	Type	Label	Description
user_id	*string*		The user to whom to grant rights. Required
rights	*Right*	repeated	The rights to grant. Required

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 `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

ListUserRightsResponse

Field	Type	Label	Description
rights	*Right*	repeated	All rights of the user.

ListUsersRequest

Required authorization: `HasRight(ParticipantAdmin)`

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

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)`

Field	Type	Label	Description
user_id	*string*		The user from whom to revoke rights. Required
rights	*Right*	repeated	The rights to revoke. Required

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 administrate 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.

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.ParticipantAdmin

The right to administrate the participant node.

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 lowercase 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

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 two 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.

Method name	Request type	Response type	Description
CreateUser	*CreateUserRequest*	*CreateUserResponse*	Create a new user. Errors: - `ALREADY_EXISTS`: if the user already exists - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields
GetUser	*GetUserRequest*	*GetUserResponse*	Get the user data of a specific user or the authenticated user. Errors: - `NOT_FOUND`: if the user doesn't exist - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields
DeleteUser	*DeleteUserRequest*	*DeleteUserResponse*	Delete an existing user and all its rights. Errors: - `NOT_FOUND`: if the user doesn't exist - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields
ListUsers	*ListUsersRequest*	*ListUsersResponse*	List all existing users. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields
GrantUserRights	*GrantUserRightsRequest*	*GrantUserRightsResponse*	Grant rights to a user. Errors: - `NOT_FOUND`: if the user doesn't exist - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields
RevokeUserRights	*RevokeUserRightsRequest*	*RevokeUserRightsResponse*	Revoke rights from a user. Errors: - `NOT_FOUND`: if the user doesn't exist - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields
ListUserRights	*ListUserRightsRequest*	*ListUserRightsResponse*	List the set of all rights granted to a user. Errors: - `NOT_FOUND`: if the user doesn't exist - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields

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. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `NOT_FOUND`: if the request does not include a valid ledger id - `FAILED_PRECONDITION`: if the ledger has been pruned after the subscription start offset - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields - `OUT_OF_RANGE`: if the absolute offset is after the end of the ledger
CompletionEnd	*CompletionEndRequest*	*CompletionEndResponse*	Returns the offset after the latest completion. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `NOT_FOUND`: if the request does not include a valid ledger id

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. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `NOT_FOUND`: if the request does not include a valid ledger id or if a resource is missing (e.g. contract key) due to for example contention on resources - `ALREADY_EXISTS` if a resource is duplicated (e.g. contract key) - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields - `ABORTED`: if the number of in-flight commands reached the maximum (if a limit is configured) - `FAILED_PRECONDITION`: on consistency errors (e.g. the contract key has changed since the submission) or if an interpretation error occurred - `UNAVAILABLE`: if the participant is not yet ready to submit commands or if the service has been shut down. - `DEADLINE_EXCEEDED`: if the request failed to receive its completion within the predefined timeout.
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. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `NOT_FOUND`: if the request does not include a valid ledger id or if a resource is missing (e.g. contract key) due to for example contention on resources - `ALREADY_EXISTS` if a resource is duplicated (e.g. contract key) - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields - `ABORTED`: if the number of in-flight commands reached the maximum (if a limit is configured) - `FAILED_PRECONDITION`: on consistency errors (e.g. the contract key has changed since the submission) or if an interpretation error occurred - `UNAVAILABLE`: if the participant is not yet ready to submit commands or if the service has been shut down. - `DEADLINE_EXCEEDED`: if the request failed to receive its completion within the predefined timeout.
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. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `NOT_FOUND`: if the request does not include a valid ledger id or if a resource is missing (e.g. contract key) due to for example contention on resources - `ALREADY_EXISTS` if a resource is duplicated (e.g. con-

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. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `NOT_FOUND`: if the request does not include a valid ledger id or if a resource is missing (e.g. contract key) due to for example contention on resources - `ALREADY_EXISTS` if a resource is duplicated (e.g. contract key) - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields - `ABORTED`: if the number of in-flight commands reached the maximum (if a limit is configured) - `FAILED_PRECONDITION`: on consistency errors (e.g. the contract key has changed since the submission) or if an interpretation error occurred - `UNAVAILABLE`: if the participant is not yet ready to submit commands or if the service has been shut down.

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.Times-		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

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

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. For example, malformed or double spend transactions will result in a `INVALID_ARGUMENT` status. Transactions with invalid time time windows (which may be valid at a later date) will result in an `ABORTED` error. 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/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 `ArchivedEvent``s, 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, if defined. Optional
create_arguments	*Record*		The arguments that have been used to create the contract. Required
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). 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

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
choice	*string*		The choice that's been exercised on the target contract. Must be a valid NameString (as described in `value.proto`). Required
choice_argument	*Value*		The argument the choice was made with. Required
acting_parties	*string*	repeated	The parties that made 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. Each element must be a valid LedgerString (as described in `value.proto`). Optional
exercise_result	*Value*		The result of exercising the choice Required

com/daml/ledger/api/v1/experimental_features.proto

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*		

ExperimentalContractIds

See *daml-lf/spec/contract-id.rst* for more information on contract ID formats.

Field	Type	Label	Description
v1	*ExperimentalContractIds.ContractIdV1Support*		

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*		

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*		

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.

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.

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. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `NOT_FOUND`: if the request does not include a valid ledger id

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.

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. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation

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. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `NOT_FOUND`: if the request does not include a valid ledger id
GetPackage	*GetPackageRequest*	*GetPackageResponse*	Returns the contents of a single package. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `NOT_FOUND`: if the requested package is unknown
GetPackageStatus	*GetPackageStatusRequest*	*GetPackageStatusResponse*	Returns the status of a single package. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `NOT_FOUND`: if the requested package is unknown

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. Errors: - `INVALID_ARGUMENT`: if `current_time` is invalid (it MUST precisely match the current time as it's known to the ledger server)

com/daml/ledger/api/v1/transaction.proto

Transaction

Filtered 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. 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

Field	Type	Label	Description
inclusive	*InclusiveFilters*		If not set, no filters will be applied. Optional

InclusiveFilters

If no internal fields are set, no filters will be applied.

Field	Type	Label	Description
template_ids	*Identifier*	repeated	A collection of templates. SHOULD NOT contain duplicates. Required

TransactionFilter

Used for filtering Transaction and Active Contract Set streams. Determines which on-ledger events will be served to the client.

Field	Type	Label	Description
filters_by_party	*TransactionFilter.FiltersByPartyEntry*	repeated	Keys of the map determine which parties' on-ledger transactions are being queried. Values of the map determine which events are disclosed in the stream per party. At the minimum, a party needs to set an empty Filters message to receive any events. Each key must be a valid PartyIdString (as described in `value.proto`). 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*		

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.

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. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `NOT_FOUND`: if the request does not include a valid ledger id - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields (e.g. if `before` is not before `end`) - `FAILED_PRECONDITION`: if the ledger has been pruned after the subscription start offset - `OUT_OF_RANGE`: if the `begin` parameter value is not before the end of the ledger
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). Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `NOT_FOUND`: if the request does not include a valid ledger id - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields (e.g. if `before` is not before `end`) - `FAILED_PRECONDITION`: if the ledger has been pruned after the subscription start offset - `OUT_OF_RANGE`: if the `begin` parameter value is not before the end of the ledger
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 Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `NOT_FOUND`: if the request does not include a valid ledger id or no such transaction exists - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields (e.g. if requesting parties are invalid or empty)
GetTransactionById	*GetTransactionByIdRequest*	*GetTransactionResponse*	Lookup a transaction tree by its ID. For looking up a transaction instead of a transaction tree, please see GetFlatTransactionById Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMISSION_DENIED`: if the claims in the token are insufficient to perform a given operation - `NOT_FOUND`: if the request does not include a valid ledger id or no such transaction exists - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields (e.g. if requesting parties are invalid or empty)
GetFlatTransactionByEven-	*GetTransactionByEventIdRequest*	*GetFlatTransactionResponse*	Lookup a transaction by the ID of an event that appears within it. Errors: - `UNAUTHENTICATED`: if the request does not include a valid access token - `PERMIS-`

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 four 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 <= 256 that match the regexp `[A-Za-z0-9:\-_ ]+`. - LedgerStrings are strings with length <= 256 that match the regexp `[A-Za-z0-9#:\-_/ ]+`. - ApplicationIdStrings are strings with length <= 256 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

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

2.2.8.5 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 {
  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*:

```
myTest = script do
    bob <- allocateParty "Bob"

    let myProduct = MyProductType with
                intField = 17
                textField = "some text"
                decimalField = 17.42
                boolField = False
                partyField = bob
```

(continues on next page)

(continued from previous page)

```
                timeField = datetime 2018 May 16 0 0 0
                listField = [1,2,3]
```

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
          int64: 1
        }
```

(continues on next page)

(continued from previous page)

```
        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 |
```

The constructor `MyConstructor1` takes a single parameter of type `Integer`, whereas the constructor `MyConstructor2` takes a record with two fields as parameter. The snippet below shows how you can create values with either of the constructors.

```
    let mySum1 = MySumConstructor1 17
```

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 12: mySum1

```
{ // Value
  variant { // Variant
    variant_id { // Identifier
      package_id: "some-hash"
      name: "Types.MySumType"
    }
    constructor: "MyConstructor1"
    value { // Value
      int64:  17
    }
  }
}
```

Listing 13: 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
```

Creating 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"
      }
    }
  }
}
```

Receiving 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"
}
```

Exercising 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
        }
      }
    }
  }
}
```

2.2.8.6 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 be written in a text or binary format. So not function types, `Update` and `Scenario` types, as well as any types built up from those.)

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`

2.2.8.7 Java bindings

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 14: Daml

```
module Foo.Bar.Baz where
```

Listing 15: 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 16: 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 17: com/acme/producttypes/Person.java

```
package com.acme.producttypes;

public class 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 18: com/acme/producttypes.Name.java

```
package com.acme.producttypes;

public class 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 19: 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 three Java classes:

1. `Bar`
2. `Bar.ContractId`
3. `Bar.Contract`

Listing 20: 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 final String owner;
  public final String name;

  public static ExerciseByKeyCommand exerciseByKeyBar_SomeChoice(BarKey key, Bar_
↪SomeChoice arg) { /* ... */ }

  public static ExerciseByKeyCommand exerciseByKeyBar_SomeChoice(BarKey key,
↪String aName) { /* ... */ }

  public CreateAndExerciseCommand createAndExerciseBar_SomeChoice(Bar_SomeChoice
↪arg) { /* ... */ }

  public CreateAndExerciseCommand createAndExerciseBar_SomeChoice(String aName) {
↪/* ... */ }

  public static class ContractId {
    public final String contractId;

    public ExerciseCommand exerciseArchive(Unit arg) { /* ... */ }

    public ExerciseCommand exerciseBar_SomeChoice(Bar_SomeChoice arg) { /* ... */
↪}

    public ExerciseCommand exerciseBar_SomeChoice(String aName) { /* ... */ }
  }

  public static class Contract {
    public final ContractId id;
    public final Bar data;

    public static Contract fromCreatedEvent(CreatedEvent event) { /* ... */ }
  }
}
```

Note that the static methods returning an `ExerciseByKeyCommand` 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 21: 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 22: com/acme/variants/BookAttribute.java

```
package com.acme.variants;

public class BookAttribute {
  public static BookAttribute fromValue(Value value) { /* ... */ }

  public static BookAttribute fromValue(Value value) { /* ... */ }
  public Value toValue() { /* ... */ }
}
```

Listing 23: 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 Value toValue() { /* ... */ }
}
```

Listing 24: com/acme/variants/bookattribute/Authors.java

```
package com.acme.variants.bookattribute;

public class Authors extends BookAttribute {
  public final List<String> listValue;

  public static Authors fromValue(Value value) { /* ... */ }

  public Author(List<String> listValue) { /* ... */ }
  public Value toValue() { /* ... */ }

}
```

Listing 25: 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 Value toValue() { /* ... */ }
}
```

Listing 26: 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 DamlRecord toValue() { /* ... */ }
}
```

Parameterized types

Note: This section is only included for completeness: we don't expect users to make use of the `fromValue` and `toValue` methods, because they 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 27: Com/Acme/ParametrizedTypes.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 28: com/acme/parametrizedtypes/Attribute.java

```
package com.acme.parametrizedtypes;

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) { /* ... */ }
}
```

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 29: Com/Acme/Enum.daml

```
module Com.Acme.Enum where

data Color = Red | Blue | Green
```

The Java code generated for this variant is:

Listing 30: com/acme/enum/Color.java

```
package com.acme.enum;


public enum Color {
  RED,

  GREEN,

  BLUE;

  /* ... */

  public static final Color fromValue(Value value$) { /* ... */ }

  public final DamlEnum toValue() {  /* ... */ }
}
```

Listing 31: com/acme/enum/bookattribute/Authors.java

```
package com.acme.enum.bookattribute;

public class Authors extends BookAttribute {
  public final List<String> listValue;
```

(continues on next page)

(continued from previous page)

```
  public static Authors fromValue(Value value) { /* ... */ }

  public Author(List<String> listValue) { /* ... */ }
  public Value toValue() { /* ... */ }

}
```

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()
            )
    );
```

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.

Setting 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.

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
│   ├── Setup.daml
│   └── Tests
│       ├── Iou.daml
│       └── Trade.daml
├── daml.yaml
├── frontend-config.js
├── pom.xml
└── src
    └── main
        ├── java
        │   └── com
        │       └── digitalasset
        │           └── 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` and `ui-backend.conf` are configuration files 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.

Overview of what an IOU is

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:

   ```
   Created .daml/dist/quickstart-0.0.1.dar.
   ```

2. To run the *sandbox* (a lightweight local version of the ledger), run `daml sandbox --dar .daml/dist/quickstart-0.0.1.dar`
   The output should look like this:

   ```
   Starting Canton sandbox.
   Listening at port 6865
   Uploading .daml/dist/quickstart-0.0.1.dar to localhost:6865
   DAR upload succeeded.
   Canton sandbox is ready.
   ```

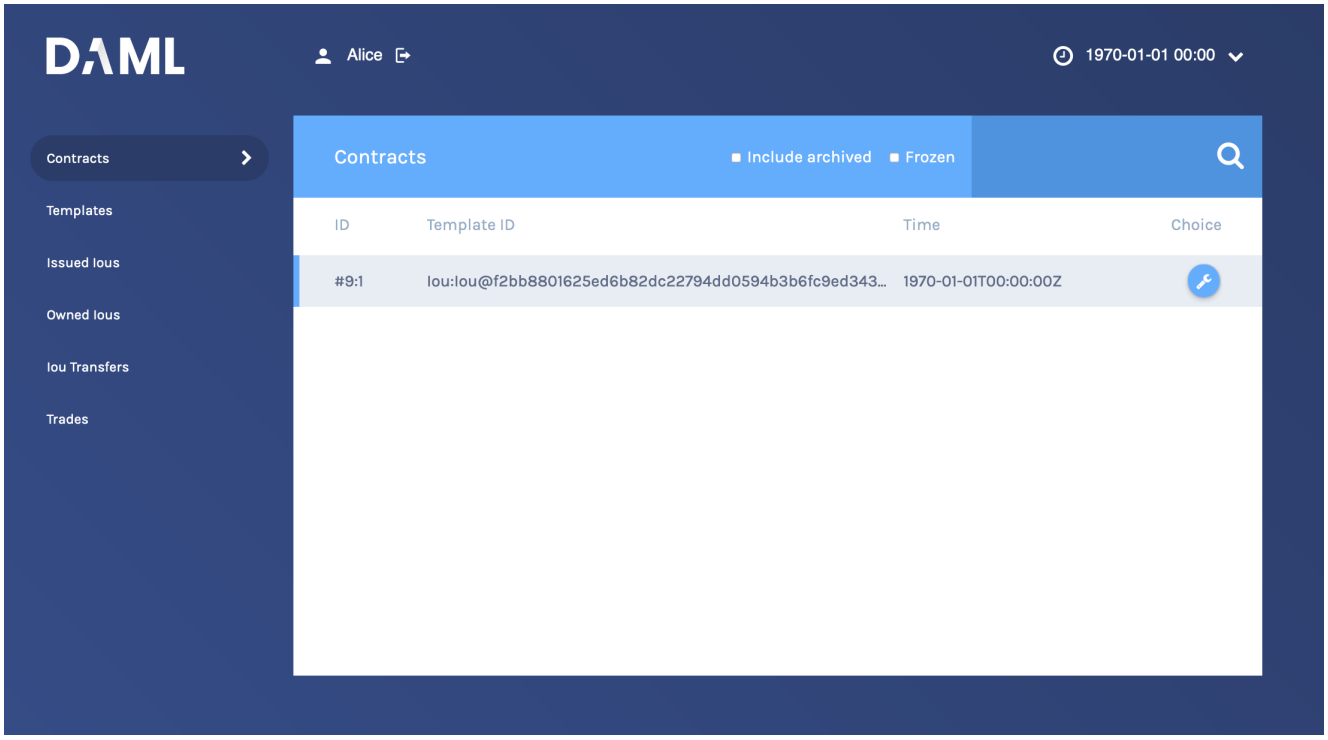
   The sandbox is now running, and you can access its *ledger API* on port `6865`.
3. Open a new terminal window and navigate to your project directory, `quickstart`.
4. To initialize the ledger with some parties and contracts we use *Daml Script* by running `daml script --dar .daml/dist/quickstart-0.0.1.dar --script-name Main:initialize --ledger-host localhost --ledger-port 6865 --static-time`
5. Start the *Navigator*, a browser-based ledger front-end, by running `daml navigator server`
   The Navigator automatically connects the sandbox. You can access it on port `4000`.

Try out the application

Now everything is running, you can try out the quickstart application:

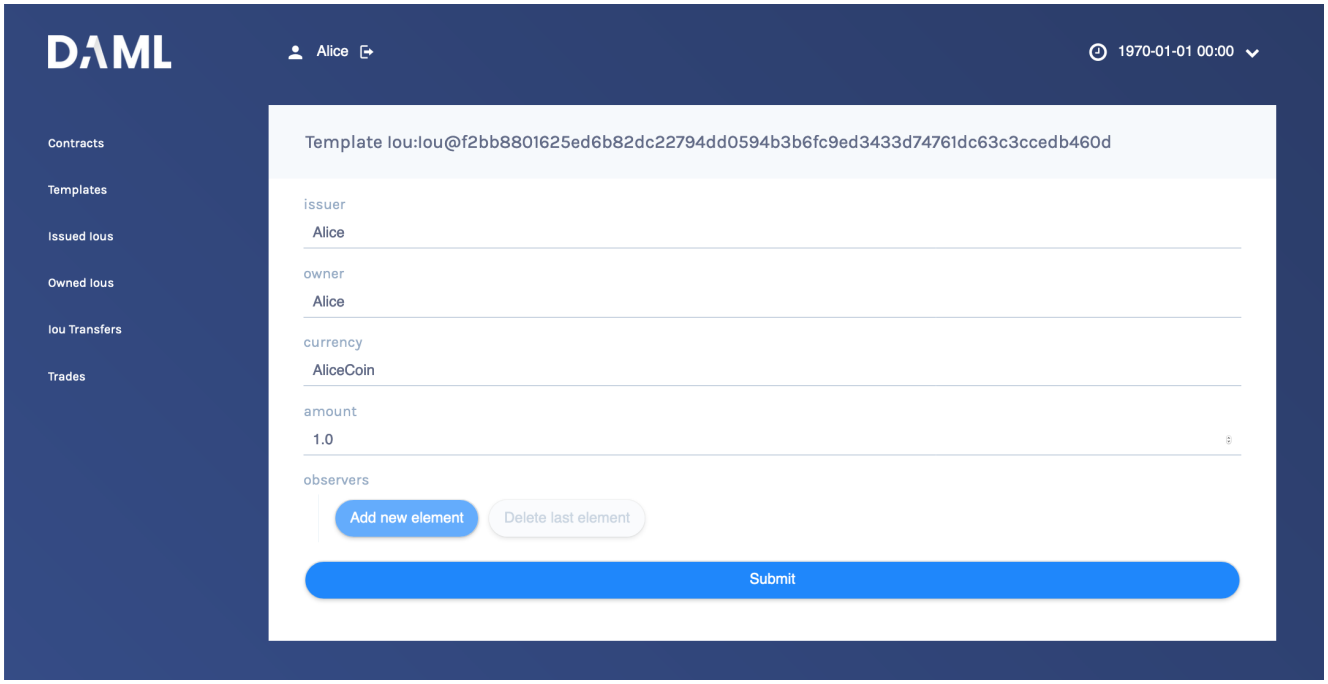
1. Go to http://localhost:4000/. This is the Navigator, which you launched *earlier*.
2. On the login screen, select **Alice** from the dropdown. This logs you in as Alice.
   (The list of available parties is specified in the `ui-backend.conf` file.)
   This takes you to the contracts view:
   This 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 `#9:1`, created from a *template* called `Iou:Iou@ffb...`.
   Your contract ID may vary. There's a lot going on in a Daml ledger, so things could have happened in a different order, or other internal ledger events might have occurred. The actual value doesn't matter. We'll refer to this contract as `#9:1` 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.
5. Try creating a contract from a template. Issue an Iou to yourself by clicking on the `Iou:Iou` row, filling it out as shown below 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**, enter *Bob* as the new owner and hit **Submit**.
8. Go to the **Owned Ious** page.
   The screen shows the same contract `#9:1` 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*. Submit *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 *#13:1*.
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.

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.
    Note its *Contract Id*.
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, enter the Contract ID you just noted 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 **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.

    USD_Bank does not know anything about the trade or the EUR-leg. For more information on privacy, refer to the *Daml Ledger Model*.

Template IouTrade:IouTrade@f2bb8801625ed6b82dc22794dd0594b3b6fc9ed343...

buyer
Alice

seller
Bob

baseIouCid
#13:1

baseIssuer
EUR_Bank

baseCurrency
EUR

baseAmount
100.0

quoteIssuer
USD_Bank

quoteCurrency
USD

quoteAmount
110.0

Submit

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 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 `100.0`, and Alice as 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`. It will not work.

Observers are specified using the `observer` keyword:

```
    observer observers
```

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
        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
```

(continues on next page)

(continued from previous page)

```
        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 to open the visualization of the resulting ledger state.

Iou:Iou

Alice	Bob	EUR_Bank	USD_Bank	id	status	issuer	owner	currency	amount	observers
X	X	-	X	#6:6	active	'USD_Bank'	'Alice'	"USD"	110.0	[]
X	X	X	-	#6:10	active	'EUR_Bank'	'Bob'	"EUR"	100.0	[]

Each row shows a contract on the ledger. The first 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 `known to (since)` show that the Banks do indeed not know anything they should not:

```
TX 6 1970-01-01T00:00:00Z (Tests.Trade:70:14)
#6:0
│   known to (since): 'Alice' (6), 'Bob' (6)
└─> 'Bob' exercises IouTrade_Accept on #5:0 (IouTrade:IouTrade)
          with
            quoteIouCid = #3:1
    children:
    #6:1
    │   known to (since): 'Alice' (6), 'Bob' (6), 'EUR_Bank' (6)
    └─> fetch #4:1 (Iou:Iou)

    #6:2
    │   known to (since): 'Alice' (6), 'Bob' (6), 'USD_Bank' (6)
    └─> fetch #3:1 (Iou:Iou)

    #6:3
    │   known 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
        │   known to (since): 'Alice' (6), 'Bob' (6), 'USD_Bank' (6)
        └─> create Iou:IouTransfer
            with
              iou =
                (Iou:Iou with
                   issuer = 'USD_Bank';
                   owner = 'Bob';
                   currency = "USD";
                   amount = 110.0000000000;
                   observers = []);
              newOwner = 'Alice'

    #6:5
    │   known to (since): 'Alice' (6), 'Bob' (6), 'USD_Bank' (6)
    └─> 'Alice' exercises IouTransfer_Accept on #6:4 (Iou:IouTransfer)
                with
        children:
        #6:6
        │   known to (since): 'Alice' (6), 'Bob' (6), 'USD_Bank' (6)
        └─> create Iou:Iou
            with
              issuer = 'USD_Bank';
              owner = 'Alice';
              currency = "USD";
              amount = 110.0000000000;
              observers = []
```

(continues on next page)

(continued from previous page)

```
#6:7
│   known 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
    │   known to (since): 'Alice' (6), 'Bob' (6), 'EUR_Bank' (6)
    └─> create Iou:IouTransfer
        with
          iou =
            (Iou:Iou with
               issuer = 'EUR_Bank';
               owner = 'Alice';
               currency = "EUR";
               amount = 100.0000000000;
               observers = ['Bob']);
          newOwner = 'Bob'

#6:9
│   known to (since): 'Alice' (6), 'Bob' (6), 'EUR_Bank' (6)
└─> 'Bob' exercises IouTransfer_Accept on #6:8 (Iou:IouTransfer)
          with
    children:
    #6:10
    │   known to (since): 'Alice' (6), 'Bob' (6), 'EUR_Bank' (6)
    └─> 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, we can now compile it using `mvn compile`.

Now start the Java integration with `mvn exec:java@run-quickstart`. Note that this step requires that the sandbox started *earlier* is running.

The application provides REST services on port `8080` to perform basic operations on behalf on `Alice`.

Note: 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`

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","owner":"Alice", "currency":"AliceCoin","amount":1.0,"observers":[]}' http://localhost:8080/iou`
`POST` on `http://localhost:8080/iou/ID/transfer` transfers the Iou with Id `ID`.
Check the Id of your new *AliceCoin* by listing all active Ious. If you have followed this guide, it will be `0` so you can run:
`curl -X POST -d '{ "newOwner":"Bob" }' 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:

```
        <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.

1. A connection to the ledger is established using a `LedgerClient` object.

```
// Create a client object to access services on the ledger.
DamlLedgerClient client = DamlLedgerClient.newBuilder(ledgerhost, ledgerport).
↪build();
```

(continues on next page)

(continued from previous page)

```
// 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 =
```

3. The Active Contracts Service (ACS) is used to quickly build up the contract store to a recent state.

```
    .getActiveContractSetClient()
    .getActiveContracts(iouFilter, true)
    .blockingForEach(
        response -> {
          response
              .getOffset()
              .ifPresent(offset -> acsOffset.set(new LedgerOffset.
↪Absolute(offset)));
          response.getCreatedEvents().stream()
              .map(Iou.Contract::fromCreatedEvent)
              .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(acsOffset.get(), iouFilter, 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()));
```

(continues on next page)

(continued from previous page)

```
                    contracts.remove(id);
                    idMap.remove(id);
                }
            }
        });
```

5. Commands are submitted via the Command Submission Service.

```
  return client
      .getCommandSubmissionClient()
      .submit(
          UUID.randomUUID().toString(),
          "IouApp",
          UUID.randomUUID().toString(),
          party,
          Optional.empty(),
          Optional.empty(),
          Optional.empty(),
          Collections.singletonList(c))
      .blockingGet();
}
```

You can find examples of `ExerciseCommand` and `CreateCommand` instantiation in the bodies of the `transfer` and `iou` endpoints, respectively.

Listing 32: ExerciseCommand

```
ExerciseCommand exerciseCommand =
    contractId.exerciseIou_Transfer(m.get("newOwner").toString());
submit(client, party, exerciseCommand);
```

Listing 33: CreateCommand

```
submit(client, party, iouCreate);
return "Iou creation submitted.";
```

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.

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`.

The Reactive Components A set of optional components you can use to assemble Daml Ledger applications. These components are deprecated as of 2020-10-14.
The most important components are:
- the `LedgerView`, which provides a local view of the Ledger
- the `Bot`, which provides utility methods to assemble automation logic for the Ledger

Can be found in the java package `com.daml.ledger.rxjava.components`.

Code generation

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*.

Connecting 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.

Getting 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.ledger</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*.

Connecting to the ledger

Before any ledger services can be accessed, a connection to the ledger must be established. This is done by creating a instance of a `DamlLedgerClient` using one of the factory methods `DamlLedgerClient.forLedgerIdAndHost` and `DamlLedgerClient.forHostWithLedgerIdDiscovery`. This instance can then be used to access service clients directly, or passed to a call to `Bot.wire` to connect a `Bot` instance to the ledger.

Authorizing

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.

Connecting 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.

Reactive Components

The Reactive Components are deprecated as of 2020-10-14.

Accessing data on the ledger: LedgerView

The `LedgerView` of an application is the copy of the ledger that the application has locally. You can query it to obtain the contracts that are active on the Ledger and not pending.

Note:

A contract is *active* if it exists in the Ledger and has not yet been archived.
A contract is *pending* if the application has sent a consuming command to the Ledger and has yet to receive an completion for the command (that is, if the command has succeeded or not).

The `LedgerView` is updated every time:

a new event is received from the Ledger
new commands are sent to the Ledger
a command has failed to be processed

For instance, if an incoming transaction is received with a create event for a contract that is relevant for the application, the application `LedgerView` is updated to contain that contract too.

Writing automations: Bot

The `Bot` is an abstraction used to write automation for a Daml Ledger. It is conceptually defined by two aspects:

the `LedgerView`
the logic that produces commands, given a `LedgerView`

When the `LedgerView` is updated, to see if the bot has new commands to submit based on the updated view, the logic of the bot is run.

The logic of the bot is a Java function from the bot's `LedgerView` to a `Flowable<CommandsAndPendingSet>`. Each `CommandsAndPendingSet` contains:

the commands to send to the Ledger
the set of contractIds that should be considered pending while the command is in-flight (that is, sent by the client but not yet processed by the Ledger)

You can wire a `Bot` to a `LedgerClient` implementation using `Bot.wire`:

```
Bot.wire(String applicationId,
         LedgerClient ledgerClient,
         TransactionFilter transactionFilter,
         Function<LedgerViewFlowable.LedgerView<R>, Flowable
↪<CommandsAndPendingSet>> bot,
         Function<CreatedContract, R> transform)
```

In the above:

`applicationId` The id used by the Ledger to identify all the queries from the same application.
`ledgerClient` The connection to the Ledger.
`transactionFilter` The server-side filter to the incoming transactions. Used to reduce the traffic between Ledger and application and make an application more efficient.

bot The logic of the application,

transform The function that, given a new contract, returns which information for that contracts are useful for the application. Can be used to reduce space used by discarding all the info not required by the application. The input to the function contains the `templateId`, the arguments of the contract created and the context of the created contract. The context contains the `workflowId`.

Example project

Example projects using the Java bindings are available on GitHub. *Read more about them here*.

2.2.8.8 Creating 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: *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

Building 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

To write an application around a Daml ledger, you'll need to interact with the **Ledger API** from another language. Every ledger that Daml can run on exposes this same API.

2.2.8.9 What's in the Ledger API

You can access the Ledger API via the HTTP JSON API, Java bindings or gRPC. In all cases, the Ledger API exposes the same 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.

2.2.8.10 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.

2.2.9 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.

2.2.9.1 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. At this time, only Daml driver for VMware Blockchain supports command deduplication across participants.

2.2.9.2 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.

2.2.10 Daml Triggers - Off-Ledger Automation in Daml

2.2.10.1 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.

Module 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.

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.

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 -> *Trigger* (TriggerState s)
Transform the high-level trigger type into the one from `Daml.Trigger.LowLevel`.

Module Daml.Trigger.Assert

Data Types

data *ACSBuilder*

Used to construct an 'ACS' for 'testRule'.

instance Monoid *ACSBuilder*

instance Semigroup *ACSBuilder*

Functions

toACS : Template 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, 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, 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.

Module 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 "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	*AnyContractId*	

instance Eq *Archived*

instance Show *Archived*

instance HasField "contractId" *Archived AnyContractId*

instance HasField "eventId" *Archived EventId*

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	AnyTemplate	

instance HasField "activeContracts" *ActiveContracts* [*Created*]

instance HasField "argument" *Created* AnyTemplate

instance HasField "contractId" *Created AnyContractId*

instance HasField "eventId" *Created EventId*

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" (*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" (*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 is (approximately) a left-fold over `Message` with an accumulator of type `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 *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" (*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" (*Trigger* s) (Party -> [Party] -> *ActiveContracts* -> *TriggerSetup* s)

instance HasField "runTriggerSetup" (*TriggerSetup* a) (Free TriggerF a)

Functions

toAnyContractId : TemplateOrInterface t => ContractId t -> *AnyContractId*
Wrap a `ContractId t` in `AnyContractId`.

fromAnyContractId : TemplateOrInterface 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*

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
```

2.2.10.2 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 applyin 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.

2.2.10.3 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 reponds 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.

2.2.10.4 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 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 types defined in the DAR. It is also possible to specify an explicit list of templates. For example, to specify just the `Message` template, one would write:

```
...
registeredTemplates = RegisteredTemplates [registeredTemplate @Message],
...
```

This is mainly useful for performance reasons if your DAR contains many templates that are not relevant for your trigger. Note that providing an explicit list of templates also filters the result of querying the ACS using the Trigger API: contracts of the excluded templates cannot be queried.

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.

2.2.10.5 Running 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.

2.2.10.6 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.

2.2.10.7 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`, we then 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 sendinga 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 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.

2.2.10.8 Command Deduplication

Daml Triggers react to many things, and it's usually important to make sure that the same command is not sent mutiple 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.

2.2.10.9 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*.

2.2.10.10 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.

2.2.11 Trigger Service

2.2.11.1 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 Usage

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);
    });
}
```

2.2.11.2 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.

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;
  }

  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 .

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 \
  --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
```

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>
  <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);
```

(continues on next page)

(continued from previous page)

```
        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.

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}
```

The *Running 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.

2.2.11.3 Starting the Trigger Service

In this example, it is assumed there is a sandbox ledger running on port 6865 on localhost.

```
daml trigger-service --config trigger-service.conf
```

where the corresponding config would be

The required config would look like

```
{
  //dar file containing the trigger
  dar-paths = [
    "./my-app.dar"
  ]

  //IP address that 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. Consider specifying `port-file` with port
↪number 0.
  port = 8088
  //port-file = "dummy-port-file"

  ledger-api {
    address = "localhost"
    port = 6865
  }

  //Optional max inbound message size in bytes. Defaults to 4194304.
  max-inbound-message-size = 4194304
  //Minimum time interval before restarting a failed trigger. Defaults to 5
↪seconds.
  min-restart-interval = 5s
  //Maximum time interval between restarting a failed trigger. Defaults to 60
↪seconds.
  max-restart-interval = 60s
  //Optional max HTTP entity upload size in bytes. Defaults to 4194304.
  max-http-entity-upload-size = 4194304
  //Optional HTTP entity upload timeout. Defaults to 60 seconds.
  http-entity-upload-timeout = 60s
  //Use static time or wall-clock, default is wall-clock time.
  time-provider-type = "wall-clock"
  //Compiler config type to use , default or dev mode
  compiler-config = "default"
  //TTL in seconds used for commands emitted by the trigger. Defaults to 30s.
  ttl = 60s

  //Initialize database and terminate.
  init-db = "false"

  //Do not abort if there are existing tables in the database schema. EXPERT ONLY.
↪ Defaults to false.
  allow-existing-schema = "false"

  trigger-store {
    user = "postgres"
```

(continues on next page)

(continued from previous page)

```
    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
  }

  authorization {
    //Sets both the internal and external auth URIs.
    //auth-common-uri = "https://oauth2/common-uri"

    // Auth Client to redirect to login , defaults to no
    auth-redirect = "yes"

    //Sets the internal auth URIs (used by the trigger service to connect
↪directly to the middleware). Overrides value set by auth-common
    auth-internal-uri = "https://oauth2/internal-uri"
    //Sets the external auth URI (the one returned to the browser). overrides
↪value set by auth-common.
    auth-external-uri = "https://oauth2/external-uri"
    //URI to the auth login flow callback endpoint `/cb`. By default constructed
↪from the incoming login request.
    auth-callback-uri =  "https://oauth2/callback-uri"

    //Optional max number of pending authorization requests. Defaults to 250.
    max-pending-authorizations = 250
    //Optional authorization timeout, defaults to 60 seconds
    authorization-timeout = 60s
  }
}
```

The above starts the Trigger Service using a number of default parameters. Most notably, the HTTP port the Trigger Service listens on which defaults to 8088. The above config file should list all available parameters, their defaults and descriptions.

The trigger-service can also be started using cli-args as shown below, one can execute the command `daml trigger-service --help` to find all available parameters.

Note: Configuration file is the recommended way to run trigger-service, running via cli-args 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
```

2.2.11.4 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 the trigger will be running as.
`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. Alice stops her running trigger like so.

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

The status endoint returns metadata about the trigger like the party it is running as and the trigger id as well as the state the trigger is in (querying the acs, running, 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

Content-Type: `application/json`
Content:

```
{ "status": "pass" }
```

2.2.12 Auth Middleware

2.2.12.1 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,
  "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:
  // 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
```

(continues on next page)

(continued from previous page)

```
  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.

Daml ledgers only validate authorization tokens. The issuance of those tokens however is something defined by the ledger operator and can vary significantly even if the same ledger is being used. This poses a challenge for application developers aiming to develop 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 ledger 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.

2.2.12.2 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 ledger operator.
Allow fine grained access control via Daml ledger claims.
Support token refresh for long running clients that should not require user interaction.

2.2.12.3 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 user is authenticated and authorized to access the given claims. The application must forward any cookies that it itself received in the user's request. The response will contain an access token and optionally a refresh token if the user 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
```

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 endoint 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.

2.3 Overview of Daml ledgers

The following table lists commercially supported Daml ledgers and environments that are available today.

Product	Synchronization Technology	Status	Canton Domain	Open Source
Daml driver for VMware Blockchain	VMware Blockchain	GA	No	No
Daml driver for PostgreSQL	PostgreSQL	GA	Yes	Yes
Daml driver for Oracle DB	Oracle DB	GA	Yes	No
Daml driver for Hyperledger Fabric	Hyperledger Fabric	Beta	Yes	No
Daml driver for Hyperledger Besu	Hyperledger Besu	Beta	Yes	No

2.3.1 Deploying 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 with an `admin` claim. If your sandbox is not setup 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

Available commands:
  list-parties             List parties known to ledger
  allocate-parties         Allocate parties on ledger
  upload-dar               Upload DAR file to ledger
  fetch-dar                Fetch DAR from ledger into file
  metering-report          Report on Ledger Use
```

2.3.1.1 Connecting 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.

2.3.1.2 Configuring 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.

2.4 Operating Daml

2.4.1 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.

2.4.1.1 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.

[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.

Command deduplication and command tracker retention should always configured in such a way, that the associated windows don't overlap with the pruning window, so 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 on existence of divulged contracts prior to or at the pruning offset. Instead, such applications MUST ensure re-divulgence of the used contracts.

2.4.1.2 How the Daml Ledger API is affected

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.

2.4.1.3 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 PII[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.

[3] This might be improved in future versions.

2.4.1.4 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:

> 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.

2.4.1.5 Determining 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.

When pruning all divulged contracts, the participant operator can choose the pruning offset 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.

2.4.2 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.

2.4.2.1 Generating 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.

Other non-metering specific Daml assistant flags may also be used alongside those shown above.

[4] Note that all the events on behalf of archived contracts and command completions found at earlier offsets will also be pruned.

2.4.2.2 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
```

2.4.2.3 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
        }
    ]
}
```

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.

2.4.3 System Requirements

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

1. An x86-compatible system running a modern Unix, Windows, or MacOS operating system.
2. Java 11 or greater.
3. An RDBMS system,

1. Either PostgreSQL 10.0 or greater.
2. Or Oracle Database 19.11 or greater.

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

1. Operating Systems:

1. Ubuntu 20.04
2. Windows Server 2016
3. MacOS 10.15 Catalina

2. Eclipse Adoptium version 11 for Java.
3. PostgreSQL 10.0
4. Oracle Database 19.11

2.4.3.1 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.

2.5 Developer Tools

2.5.1 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.

2.5.1.1 Full help for commands

To see information about any command, run it with `--help`.

2.5.1.2 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
```

(continues on next page)

(continued from previous page)

```
  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.

2.5.1.3 Building 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
```

2.5.1.4 Managing 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.

2.5.1.5 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`.

2.5.1.6 Running 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 usecase, 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.

2.5.2 Daml Studio

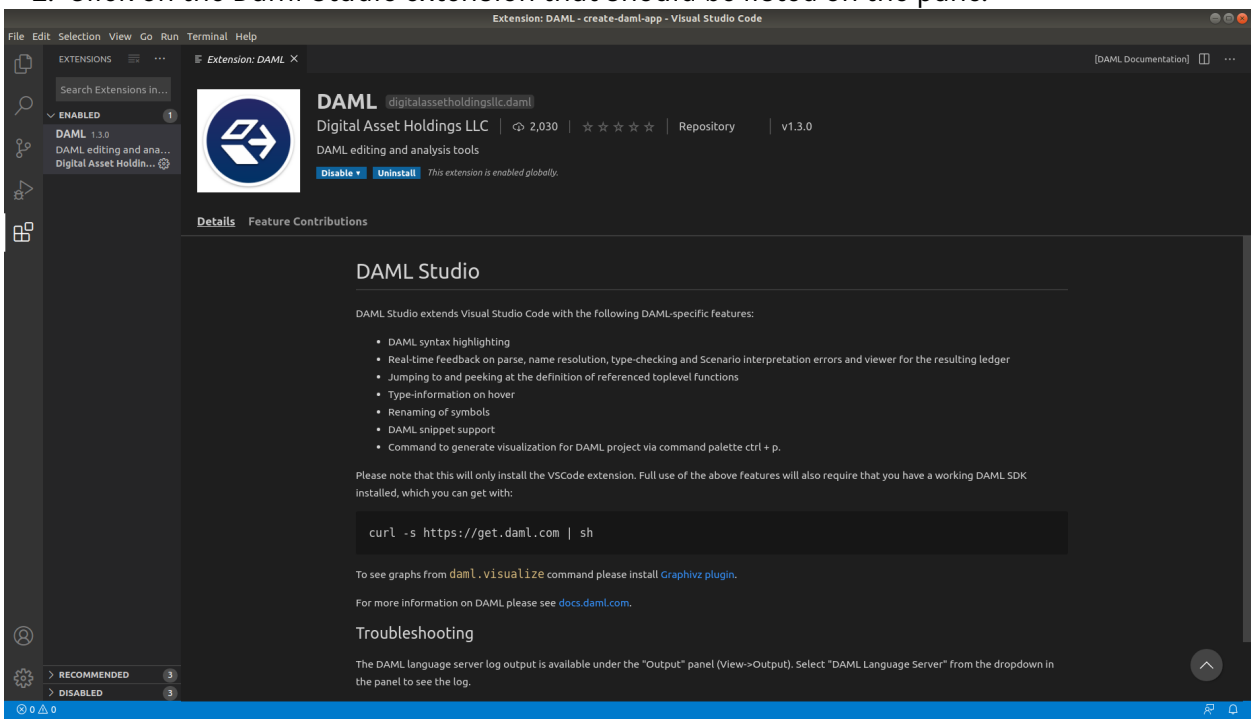
Daml Studio is an integrated development environment (IDE) for Daml. It is an extension on top of Visual Studio Code (VS Code), a cross-platform, open-source editor providing a rich code editing experience.

2.5.2.1 Installing

Daml Studio is included in *the Daml SDK*.

2.5.2.2 Creating your first Daml file

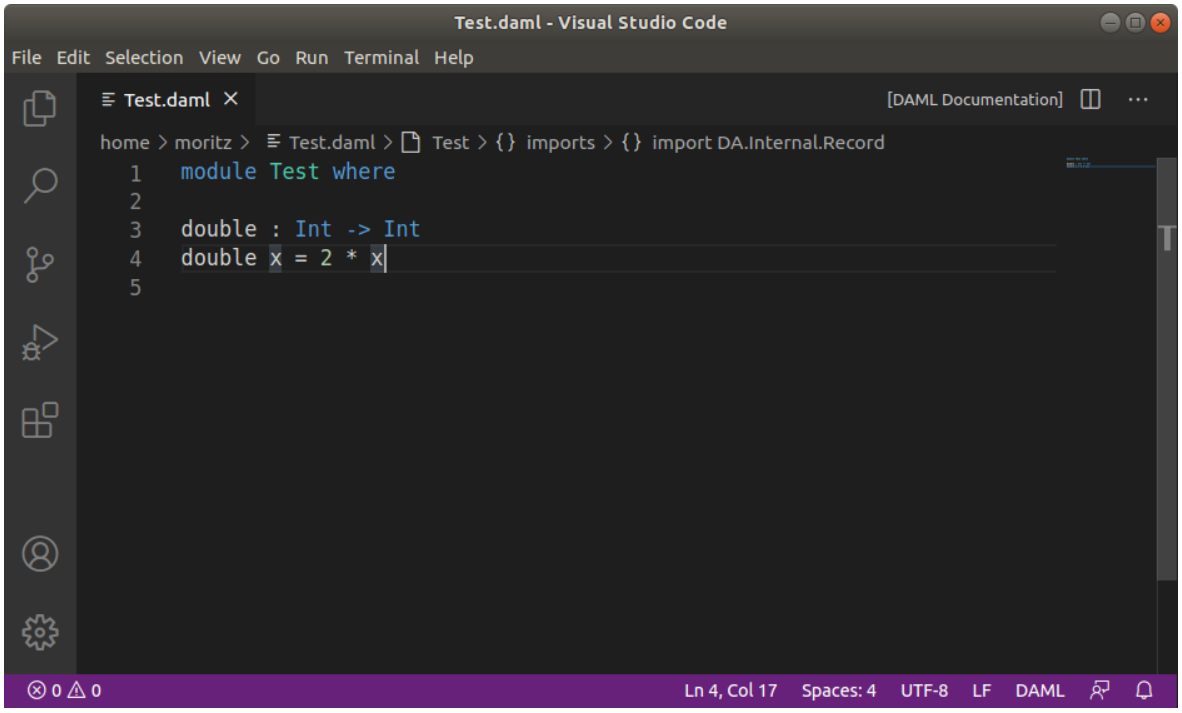
1. Start Daml Studio by running `daml studio` in the current project.
   This command starts Visual Studio Code and (if needs be) installs the Daml Studio extension, or upgrades it to the latest version.
2. Make sure the Daml Studio extension is installed:
   1. Click on the Extensions icon at the bottom of the VS Code sidebar.
   2. Click on the Daml Studio extension that should be listed on the pane.



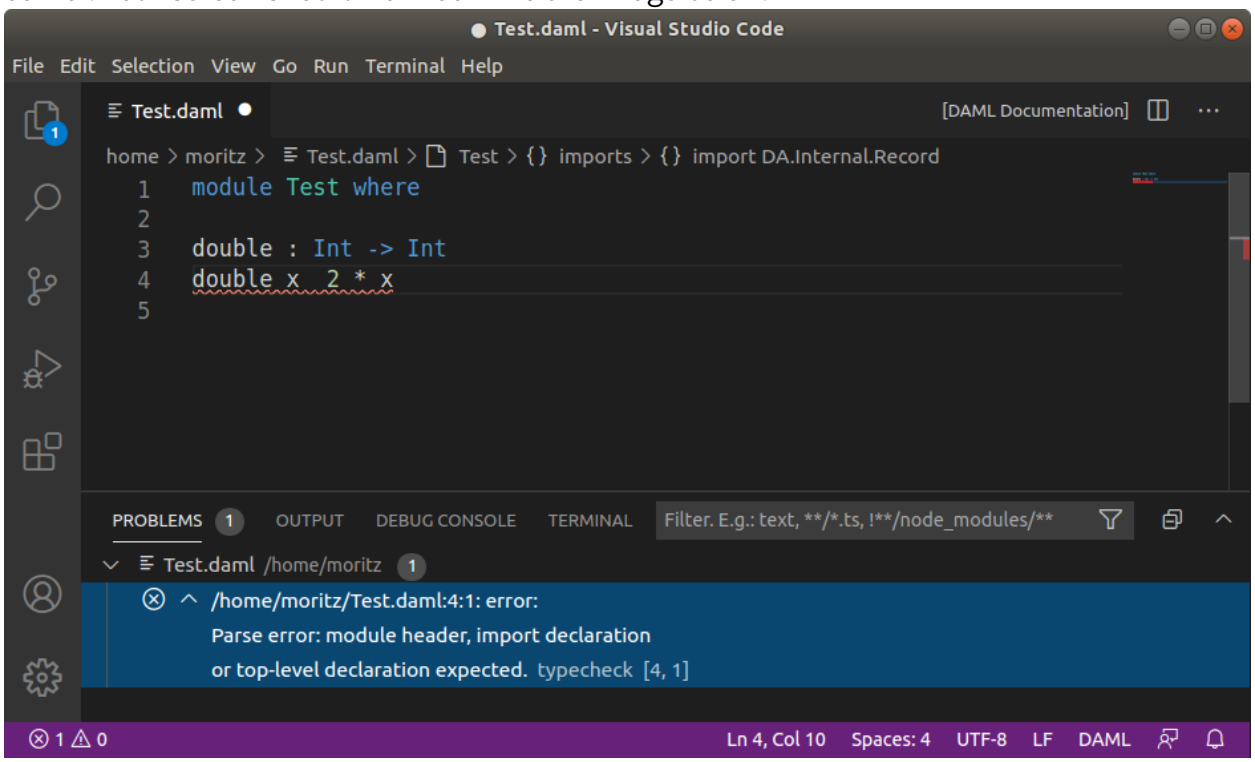
3. Open a new file (⌘N) and save it (⌘S) as `Test.daml`.
4. Copy the following code into your file:

```
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 docs*.

2.5.2.3 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

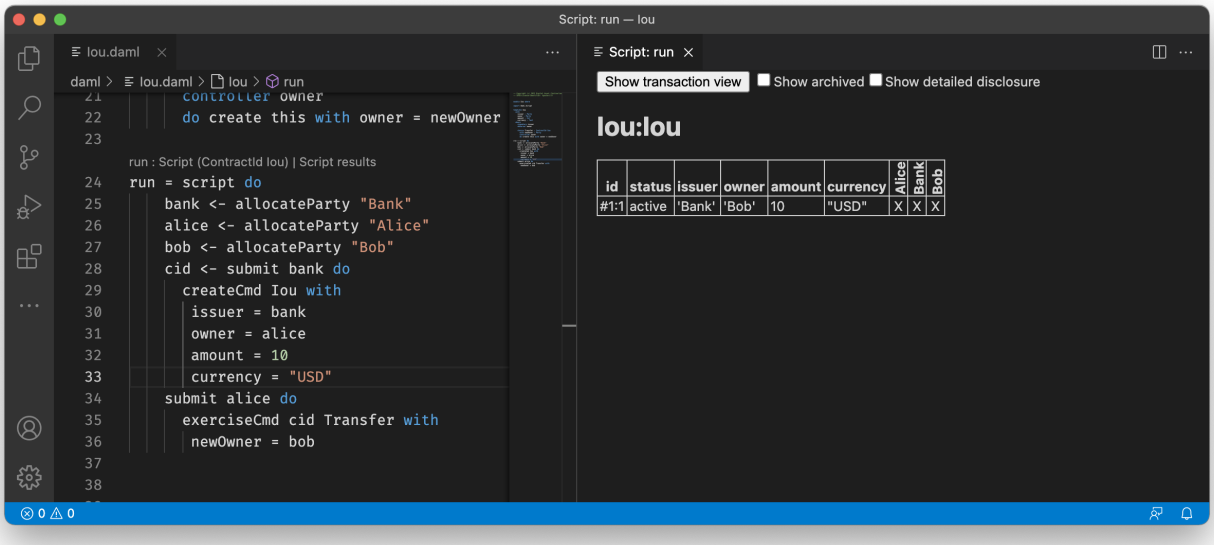
Hover tooltips

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.

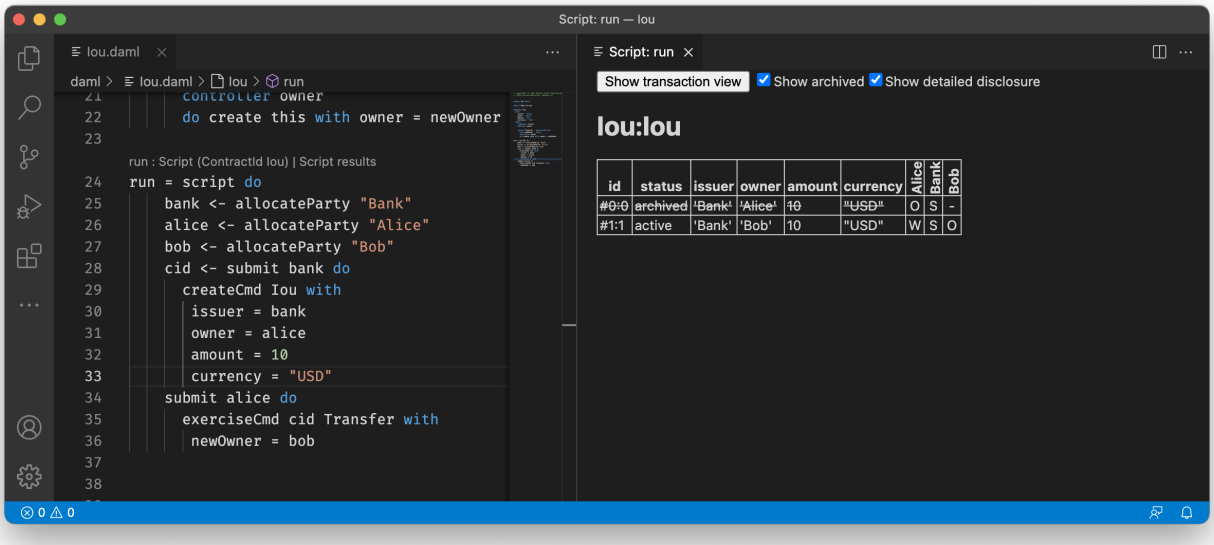


If you want more details, you can click on the *Show archived* checkbox, which extends the table to include archived contracts, and on the *Show detailed disclosure* checkbox, which displays why the contract is visible to each party, based on four categories:

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:

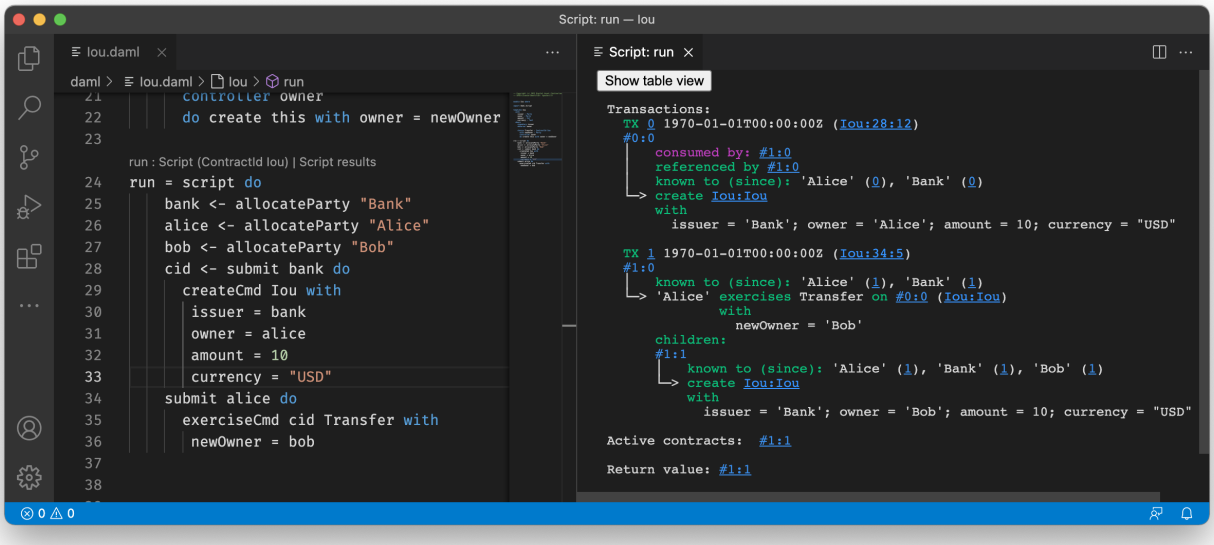


Fig. 8: Script results

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`).

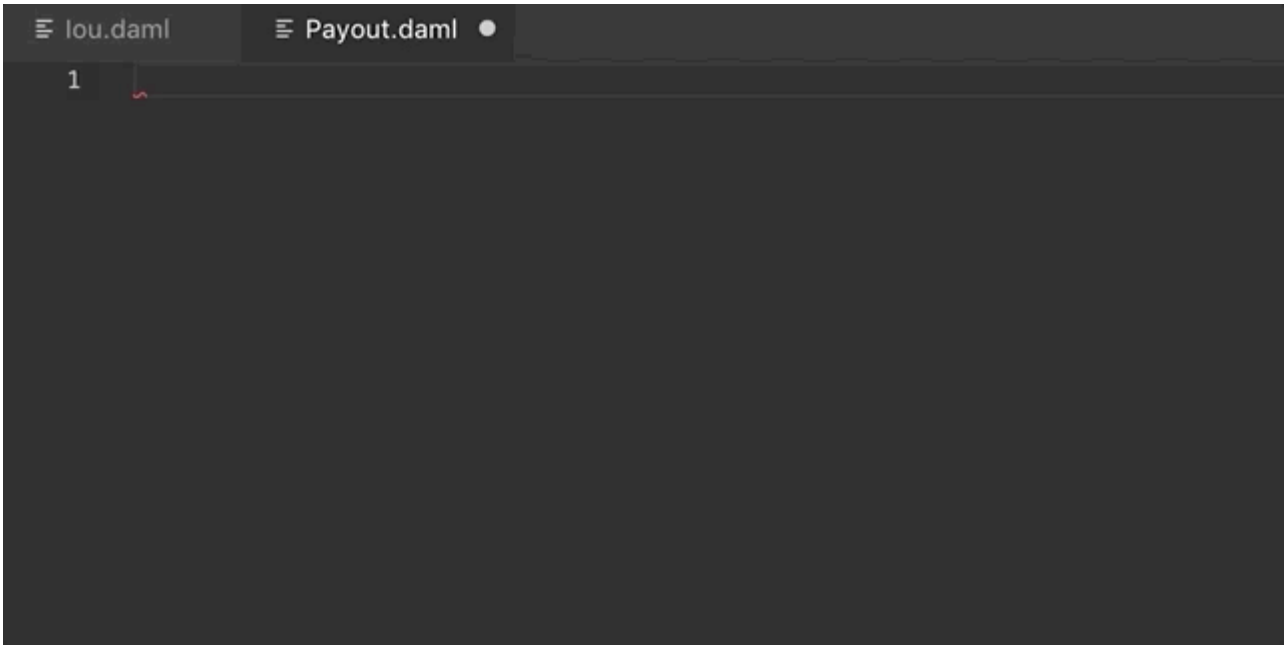
Daml snippets

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.

2.5.2.4 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"
```

```
Scenario 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:

```
Scenario 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
     └─> create 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

```
Scenario 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
  │   known to (since): 'Alice' (#0), 'Bob' (#0)
  └─> create 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:

```
Scenario 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
  │   known to (since): 'Alice' (#0)
  └─> create NotVisibleFailure:Example
      with
        owner = 'Alice'
```

To fix this issue the party 'Bob' should be made a controlling party in one of the choices.

2.5.2.5 Working 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.

2.5.3 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
```

2.5.3.1 Running 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 34: auth.conf

```
canton.participants.sandbox.ledger-api.auth-services = [{
    // type can be
    //   jwt-rs-256-crt
    //   jwt-es-256-crt
    //   jwt-es-512-crt
    type = jwt-rs-256-crt
    certificate = my-certificate.cert
}]
```

`jwt-rs-256-crt`. The sandbox 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 sandbox 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 sandbox 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 a JWKS URL. In that case, the sandbox will expect all tokens to be signed with RS256 (RSA Signature with SHA-256) with the public key loaded from the given JWKS URL.

Listing 35: auth.conf

```
canton.participants.sandbox.ledger-api.auth-services = [{
    type = jwt-rs-256-jwks
    url = "https://path.to/jwks.key"
}]
```

Warning: For testing purposes only, you can also specify a shared secret. In that case, the sandbox will expect all tokens to be signed with HMAC256 with the given plaintext secret. This is not considered safe for production.

Listing 36: auth.conf

```
canton.participants.sandbox.ledger-api.auth-services = [{
    type = unsafe-jwt-hmac-256
    secret = "not-safe-for-production"
}]
```

Generating JSON Web Tokens (JWT)

To generate access tokens for testing purposes, use the jwt.io web site.

Generating 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

Generating 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
```

2.5.3.2 Running 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. Below, you can see an example config. For more details on TLS, refer to *Canton's documentation on TLS*.

Listing 37: 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"
    // 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"
      }
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

2.5.3.3 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`.

2.5.3.4 Metrics

Enable and configure reporting

You can enable metrics reporting via Prometheus using the following configuration file.

Listing 38: 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.deduplicated_commands`

A meter. Number of deduplicated commands.

`daml.commands.delayed_submissions`

A meter. Number of delayed submissions (submission who have been evaluated to transaction with a ledger time farther in the future than the expected latency).

`daml.commands.failed_command_interpretation`

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.sandbox.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.sandbox.pool.Wait` (timer): time spent waiting to acquire a connection
`daml.index.db.connection.sandbox.pool.Usage` (histogram): time spent using each acquired connection
`daml.index.db.connection.sandbox.pool.TotalConnections` (gauge): number or total connections
`daml.index.db.connection.sandbox.pool.IdleConnections` (gauge): number of idle connections

`daml.index.db.connection.sandbox.pool.ActiveConnections` (gauge): number of active connections
`daml.index.db.connection.sandbox.pool.PendingConnections` (gauge): number of threads waiting for a connection

daml.index.db.deduplicate_command

A timer. Time spent persisting deduplication information to ensure the continued working of the deduplication mechanism across restarts.

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.remove_expired_deduplication_data

A database metric. Time spent removing deduplication information after the expiration of the deduplication window. Deduplication information is persisted to ensure the continued working of the deduplication mechanism across restarts.

daml.index.db.stop_deduplicating_command

A database metric. Time spent removing deduplication information after the failure of a command. Deduplication information is persisted to ensure the continued working of the deduplication mechanism across restarts.

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.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.

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.

2.5.4 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.

2.5.4.1 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.)

2.5.4.2 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.

2.5.4.3 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:

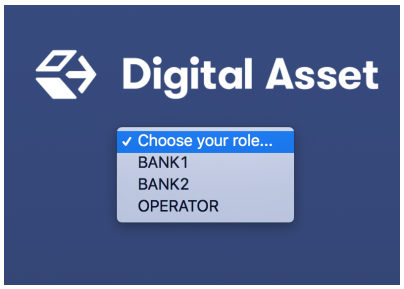
```
alice <- allocatePartyWithHint "Alice" (PartyIdHint "Alice")
bob <- allocatePartyWithHint "Bob" (PartyIdHint "Bob")
aliceId <- validateUserId "alice"
```

(continues on next page)

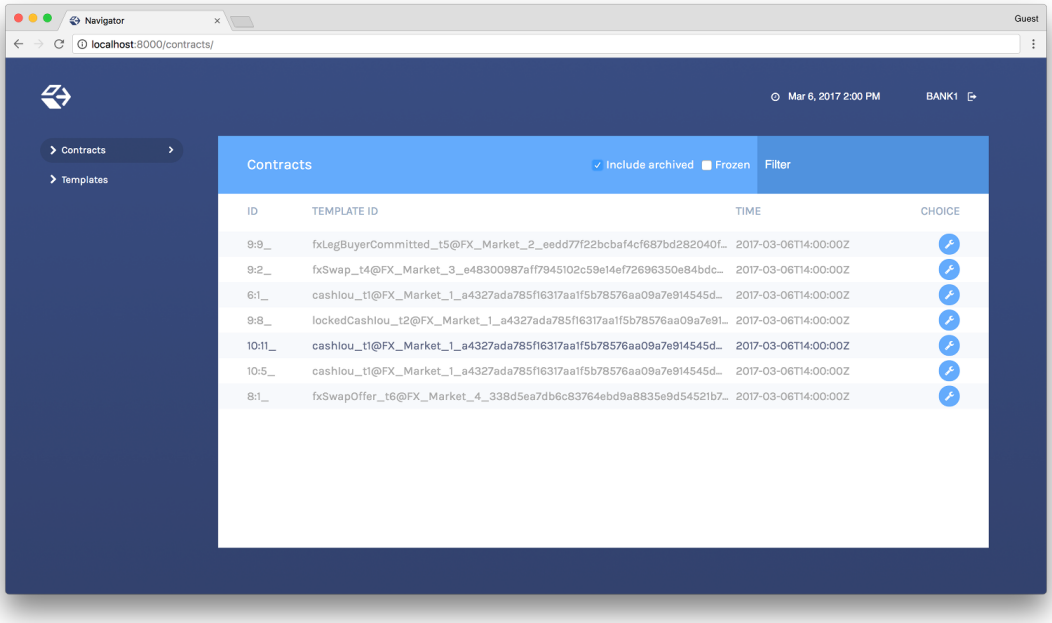
(continued from previous page)

```
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 drop-down list on the Navigator home screen to select from the available users.

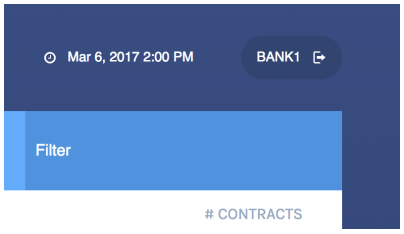


The main Navigator screen will be displayed, with contracts that the primary party of this user is entitled to view in the main pane and the option to switch from contracts to templates in the pane at the left. Other options allow you to filter the display, include or exclude archived contracts, and exercise choices as described below.



To change the active user:

1. Click the name of the current user in the top right 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
```

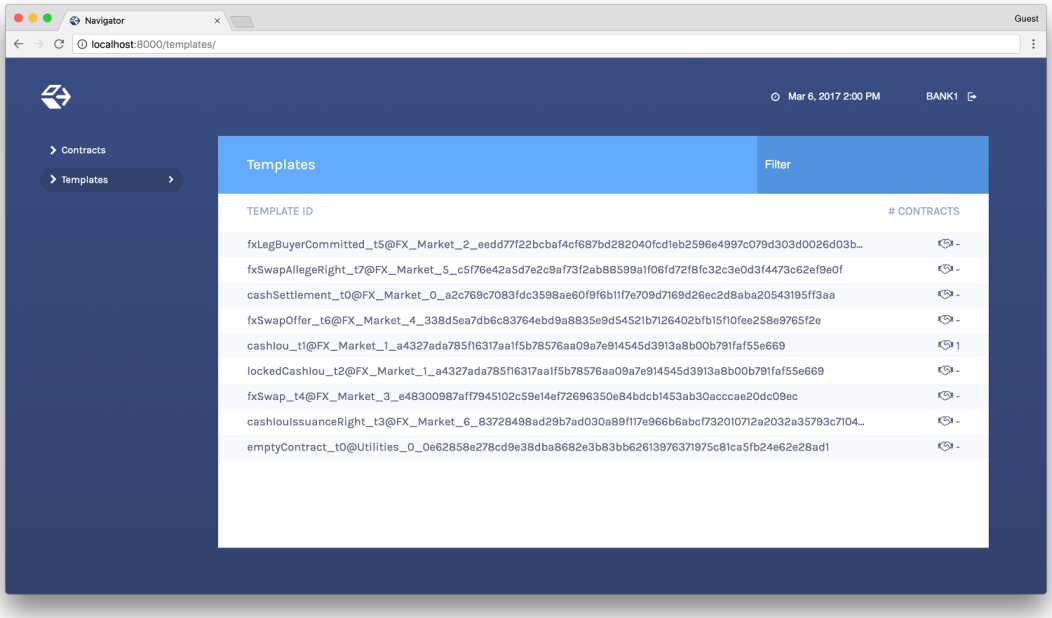
2.5.4.4 Viewing templates or contracts

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

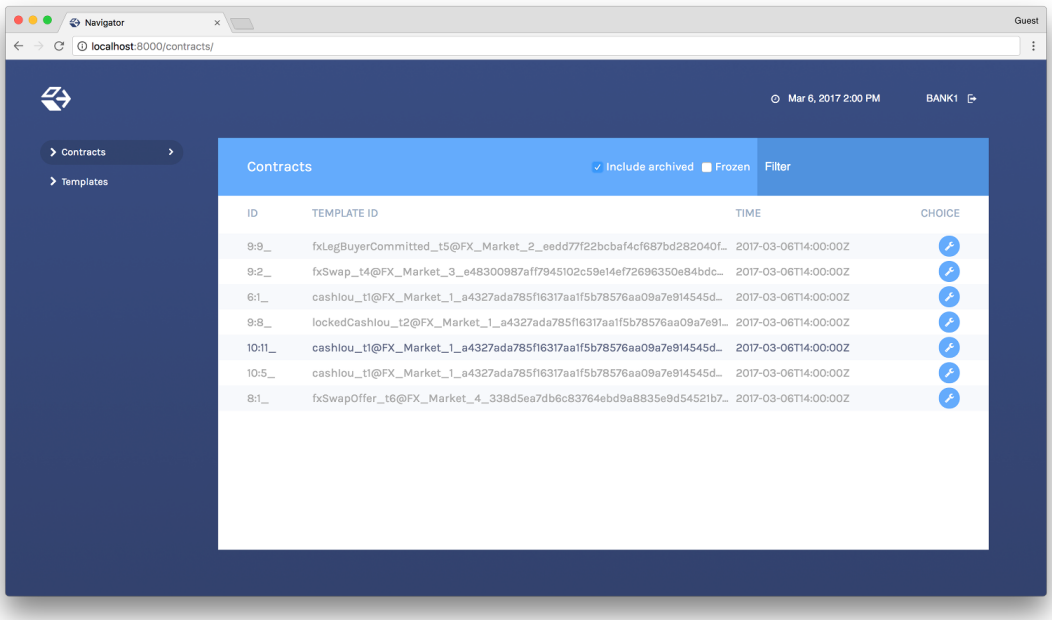
To see what contract templates are available on the ledger you are connected to, choose **Templates** in the left pane of the main Navigator screen.



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.

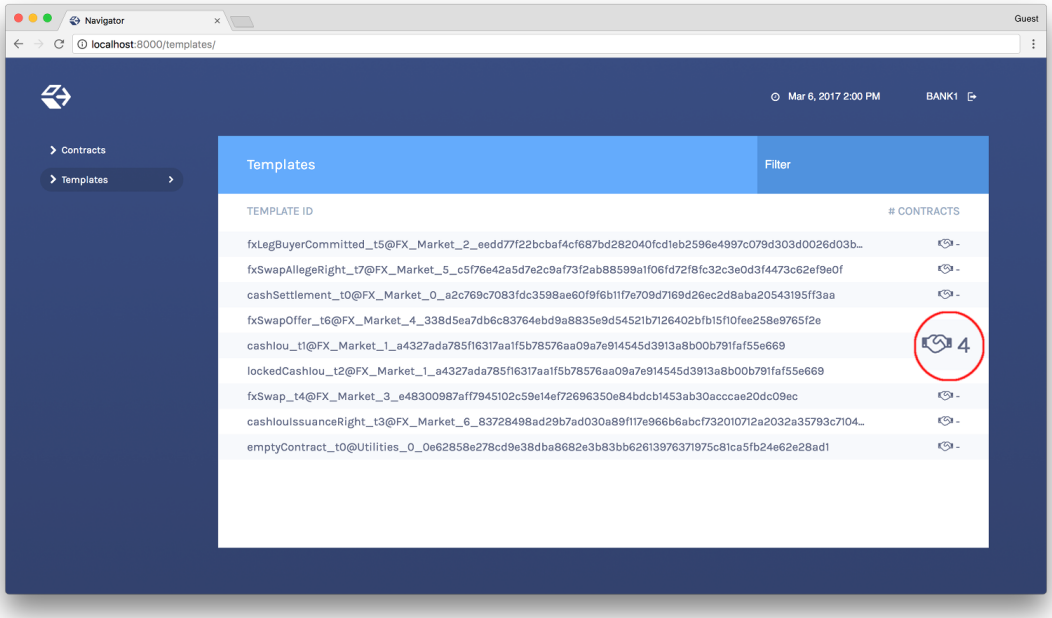
Use the **Include Archived** checkbox at the top to include or exclude archived contracts.

Viewing contracts based on a template

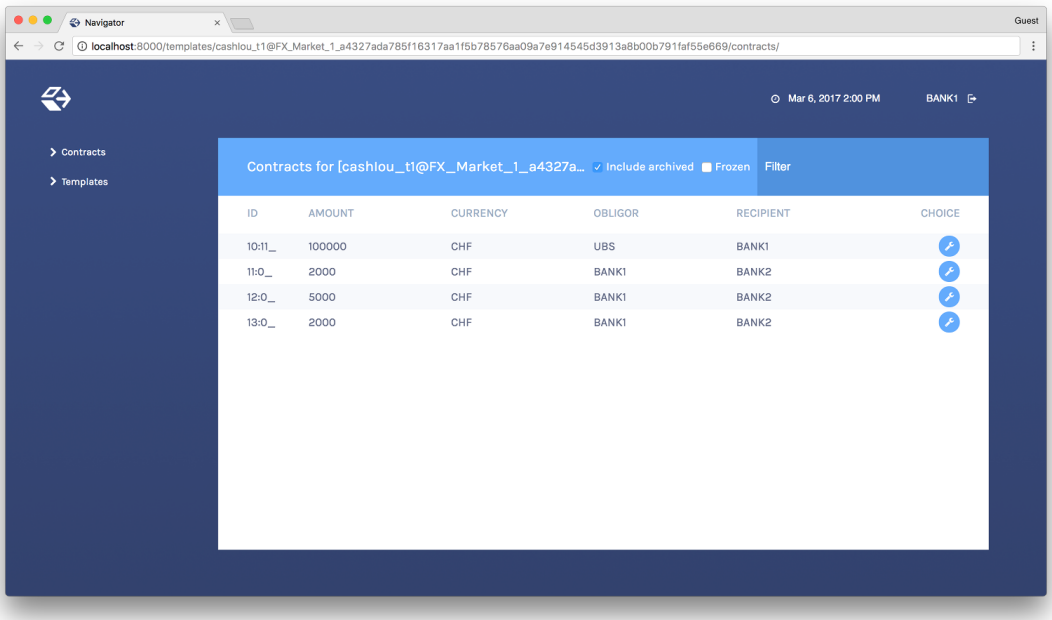
You can also view the list of contracts that are based on a particular template.

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



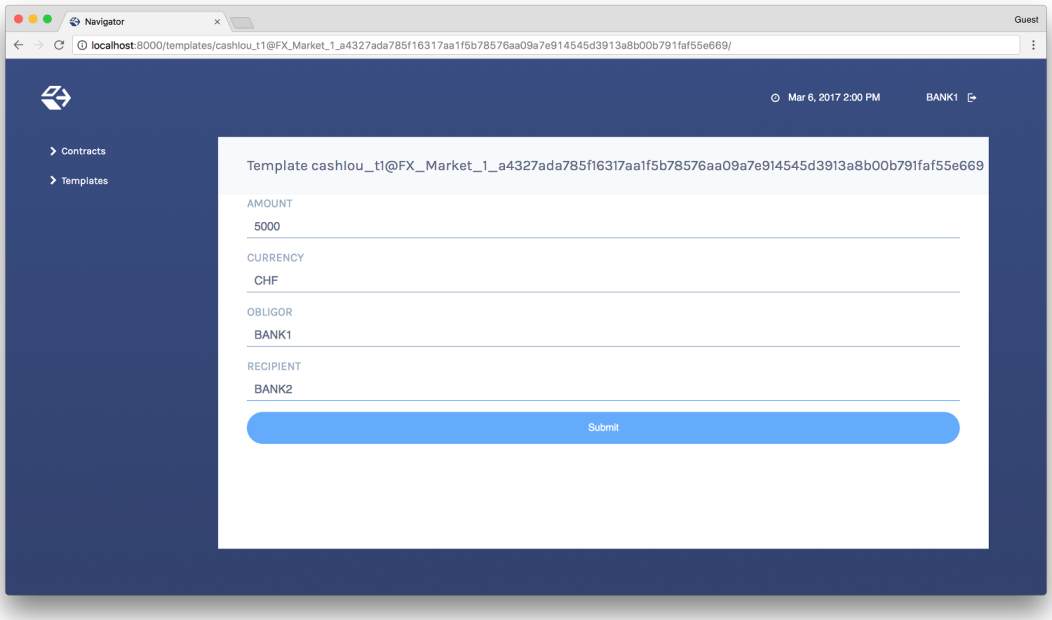
List of Contracts



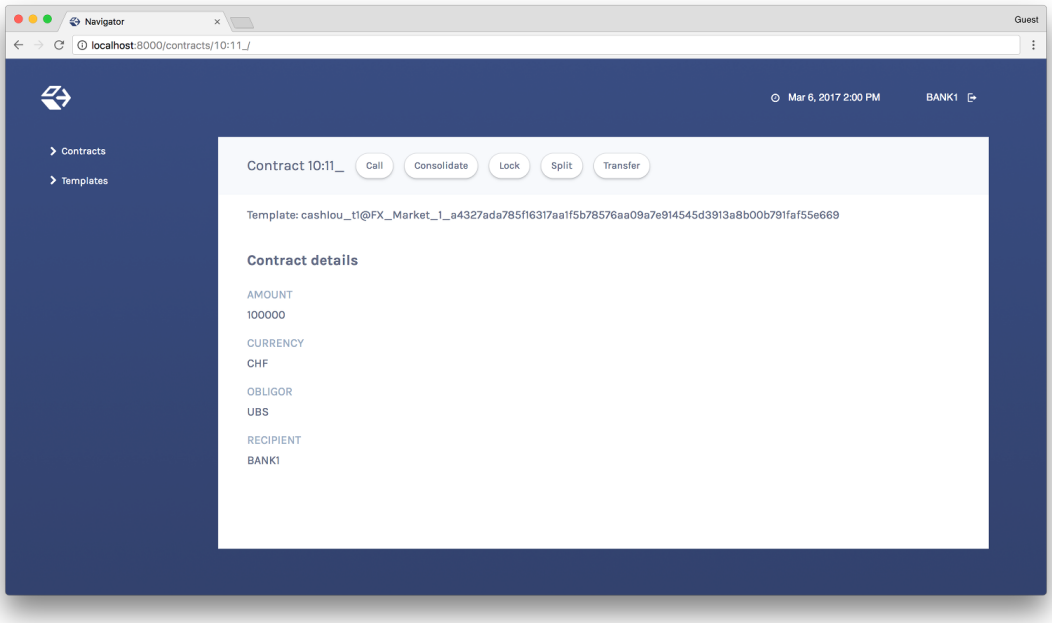
Viewing template and contract details

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



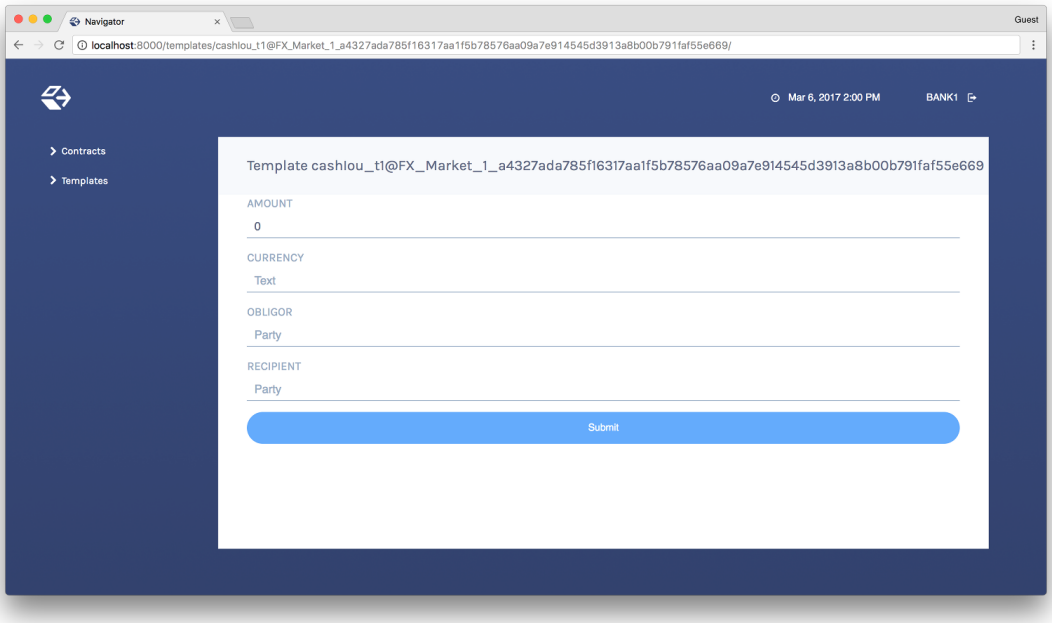
2.5.4.5 Using Navigator

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.

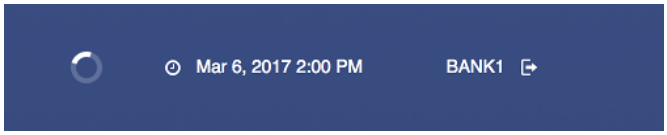
To create a contract based on a template:

1. Navigate to the template detail page as described above.
2. Complete the values in the form
3. Choose the **Submit** button.

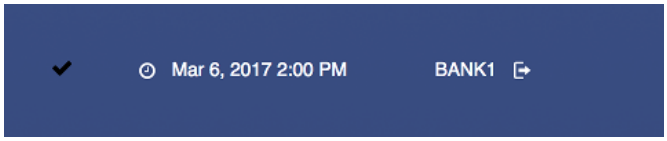


When the command has been committed to the ledger, the loading indicator in the navbar at the top will display a tick mark.

While loading



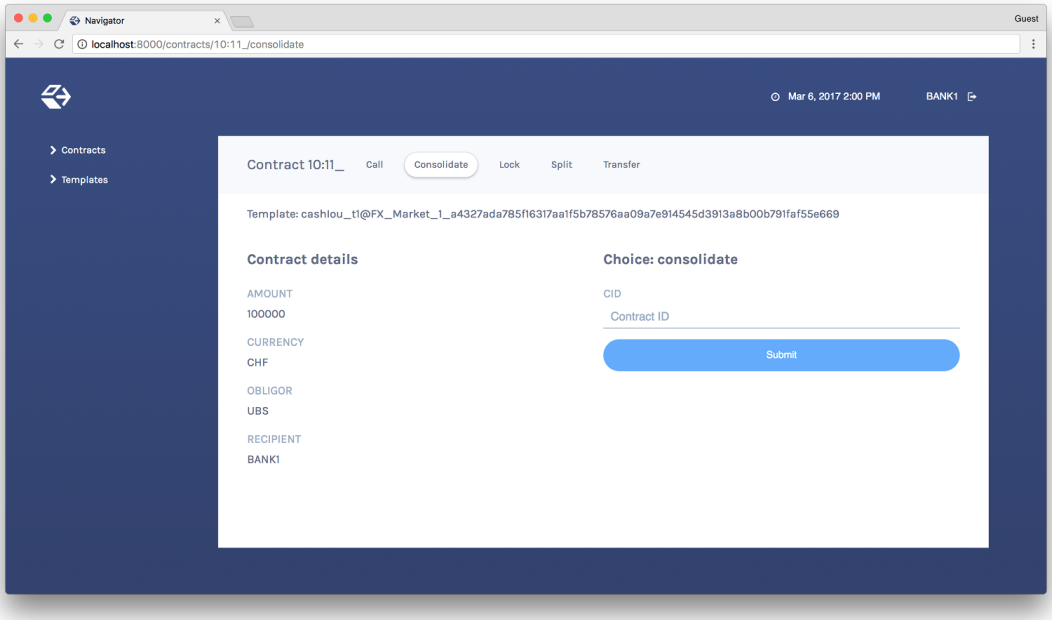
When committed to the ledger



Exercising choices

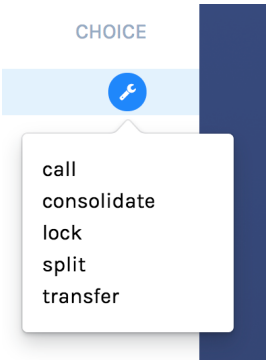
To exercise a choice:

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.



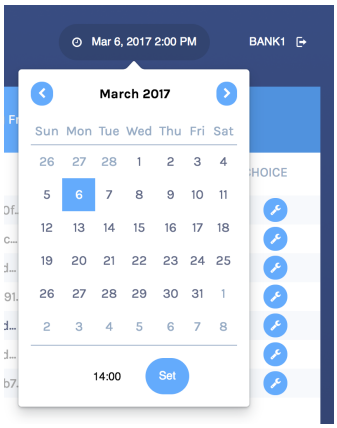
You will see the loading and confirmation indicators, as pictured above in Creating Contracts.

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.



2.5.4.6 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:

2.5.4.7 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
```

2.5.5 Daml codegen

2.5.5.1 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`.

2.5.5.2 Running 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
-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:
```

(continues on next page)

(continued from previous page)

```
    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

2.5.6 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.

2.5.6.1 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 39: 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
```

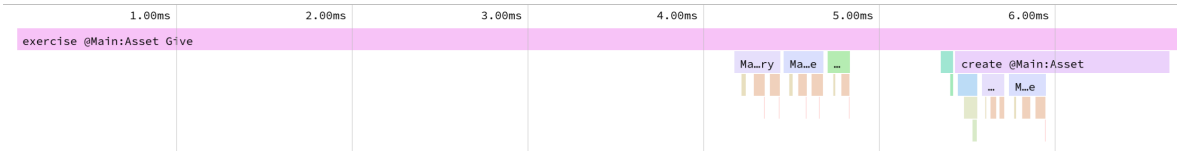
If we now look at the contents of the `profile-results` directory, we can see one JSON file per transaction produced by the script. Each file has a name of the form `$timestamp-$command.json` where `$timestamp` is the submission time of the transaction and `$command` is a human-readable description of 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.

2.5.6.2 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
```

Chapter 3

Canton Guide

3.1 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.

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*.

3.2 Tutorials

Tutorials

To understand what Canton offers and how it is different to existing solutions, run or watch the *reference demo*.
To learn the basic concepts of Canton, use our *getting started* tutorial and use our advanced configuration examples which ship with the packaged release.
To develop applications using Canton, follow the *Create Daml App guide*.
Follow our *installation guide* to install your Canton nodes.

Background Concepts

In order to get an overview and to clarify terminology, consult the Glossary of Concepts section.
To understand how Canton works in detail and what requirements it fulfills, consult the *architecture manual*.
To dive deeply into the theory, read the Daml Ledger model, as Canton implements this model faithfully.

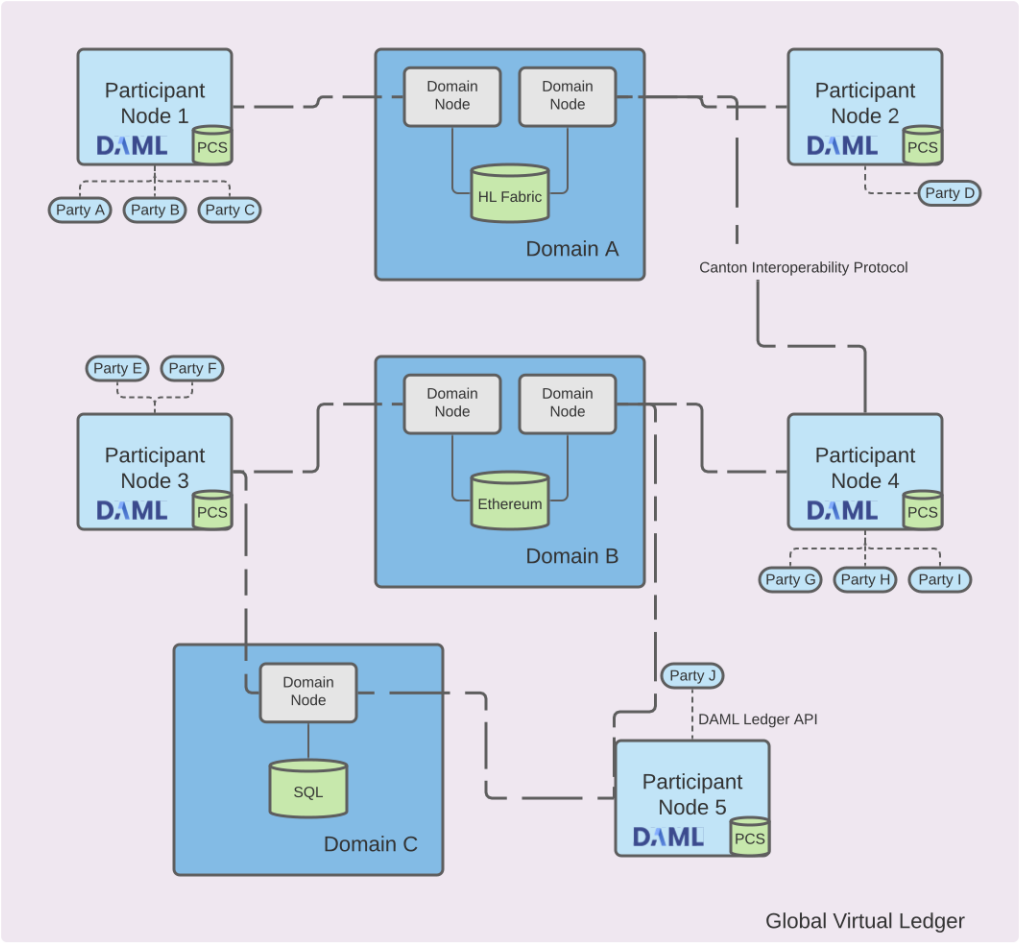


Fig. 1: Parties are hosted on participant nodes. Applications connect as parties to their participant node using the Ledger API. The participant node runs the Daml interpreter for the locally installed Daml smart contract code and stores the smart contracts in the *private contract store (PCS)*. The 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.

3.2.1 Canton Demo

Note: The Canton demo currently works only with the enterprise version of the Canton ledger. With the Canton community edition, the demo breaks in the GDPR part. Community users can watch the video recording below to get an idea of this part.

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-x.y.z.tar.gz
cd canton-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`.

3.2.2 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

3.2.2.1 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*).

Otherwise Canton is platform-agnostic, but we recommend you try it under Linux and macOS if possible as we currently only test those platforms. Under Windows, the Canton console output will be garbled 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 our 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
├── deployment
├── 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
`deployment`: contains a few example deployments to cloud or docker
`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.

3.2.2.2 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. However, as it's built using Ammonite, you also have the full power of Scala if you need to extend it with new scripts.

```
@ 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. 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
..
```

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
```

```
@ participant1.help("start")
start
Start the instance
```

3.2.2.3 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
```

(continues on next page)

(continued from previous page)

```
      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 a single 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:

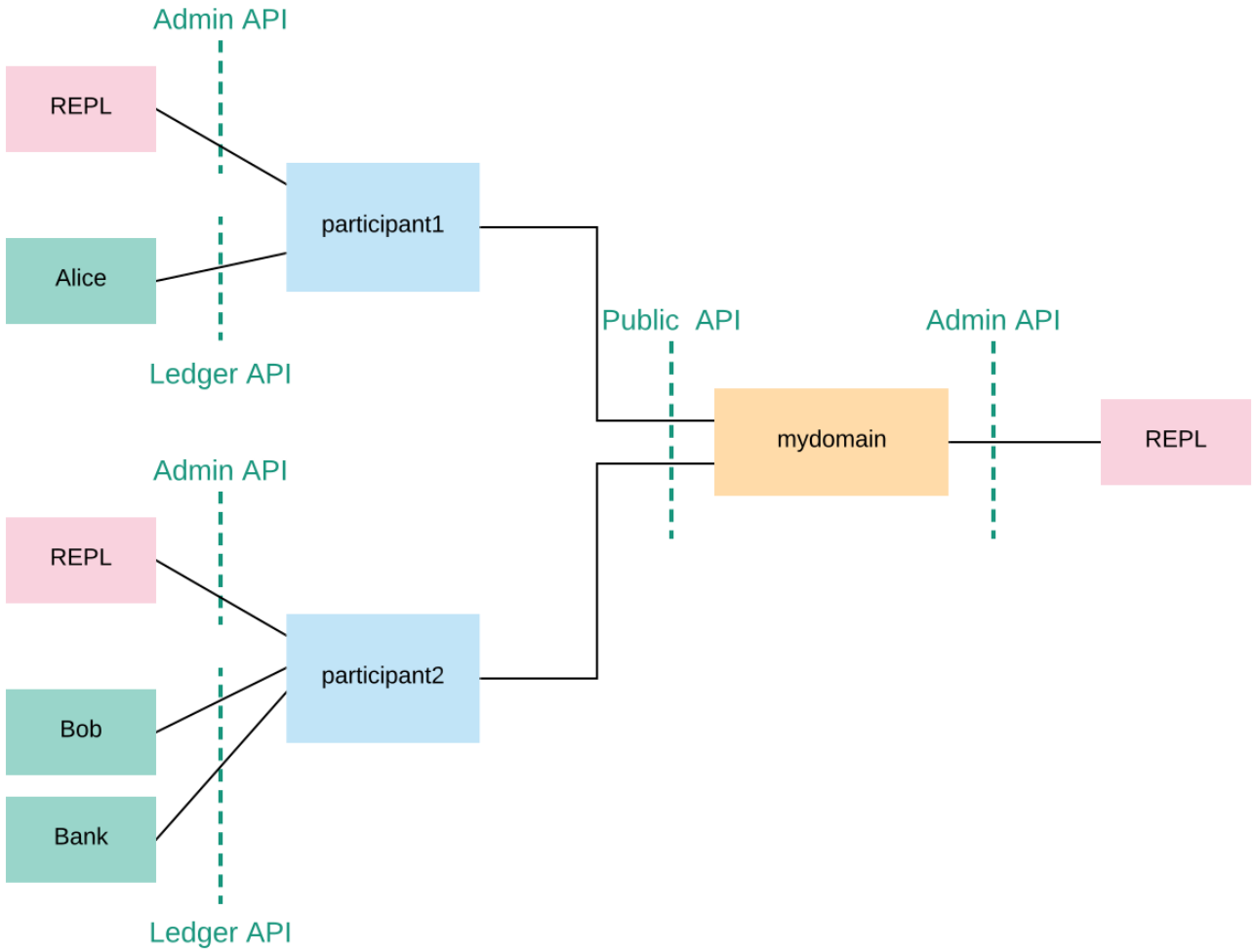
manage the participant node's connections to domains
add or remove parties to be hosted at the participant node
upload new Daml archives
configure the operational data of the participant, such as cryptographic keys
run diagnostic commands

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


Admin API
REPL
participant1
Alice
Ledger API
Public API
Admin API
mydomain
REPL
Admin API
REPL
participant2
Bob
Bank
Ledger API

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.

3.2.2.4 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::1220ff0ca3d6b6d27f519f44ed1a9c31cd123e1d19e0fa0c5c7227cd1ceaefd1ba96
Uptime: 4.593125s
Ports:
    admin: 15023
    public: 15022
Connected Participants: None
Sequencer: SequencerHealthStatus(isActive = true)

Status for Participant 'participant1':
Participant id:
↪PAR::participant1::1220ef59f36b4a1a62b33eb463d67bfb5c7835151e8ef6e7b8774847dfb9daba5c4e
Uptime: 3.71267s
Ports:
    ledger: 15018
    admin: 15019
Connected domains: None
Unhealthy domains: None
Active: true

Status for Participant 'participant2':
Participant id:
↪PAR::participant2::12209795b95ead3212d0d792d953860a30686cb78e04fa6af7e968db894c13fe6481
Uptime: 2.654984s
Ports:
    ledger: 15020
    admin: 15021
Connected domains: None
Unhealthy domains: None
Active: true
```

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::1220ef59f36b4a1a62b33eb463d67bfb5c7835151e8ef6e7b8774847dfb9daba5c4e
Uptime: 3.911613s
Ports:
    ledger: 15018
    admin: 15019
Connected domains: None
```

(continues on next page)

(continued from previous page)

```
Unhealthy domains: None
Active: true
```

or for the domain:

```
@ mydomain.health.status
res7: com.digitalasset.canton.health.admin.data.NodeStatus[com.digitalasset.
↪canton.health.admin.data.DomainStatus] = Domain id:
↪mydomain::1220ff0ca3d6b6d27f519f44ed1a9c31cd123e1d19e0fa0c5c7227cd1ceaefd1ba96
Uptime: 5.072006s
Ports:
    admin: 15023
    public: 15022
Connected Participants: None
Sequencer: SequencerHealthStatus(isActive = true)
```

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::1220ff0ca3d6b6d27f519f44ed1a9c31cd123e1d19e0fa0c5c7227cd1ceaefd1ba96
Uptime: 7.864291s
Ports:
    admin: 15023
    public: 15022
Connected Participants:
    PAR::participant2::12209795b95e...
    PAR::participant1::1220ef59f36b...
Sequencer: SequencerHealthStatus(isActive = true)

Status for Participant 'participant1':
Participant id:
↪PAR::participant1::1220ef59f36b4a1a62b33eb463d67bfb5c7835151e8ef6e7b8774847dfb9daba5c4e
Uptime: 6.9292s
Ports:
..
```

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: concurrent.duration.Duration = 516 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.

3.2.2.5 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: com.digitalasset.canton.DomainId = mydomain::1220ff0ca3d6...
```

```
@ participant1.id
res13: ParticipantId = PAR::participant1::1220ef59f36b...
```

```
@ participant2.id
res14: ParticipantId = PAR::participant2::12209795b95e...
```

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).

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::1220ef59f36b...
```

```
@ val bob = participant2.parties.enable("Bob")
bob : PartyId = Bob::12209795b95e...
```

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::1220ef59f36b...,
    participants = Vector(
      ParticipantDomains(
        participant = PAR::participant1::1220ef59f36b...,
        domains = Vector(
          DomainPermission(domain = mydomain::1220ff0ca3d6..., permission =
↪Submission)
        )
      )
    )
  )
)
```

and the same for Bob:

```
@ mydomain.parties.list("Bob")
res18: Seq[ListPartiesResult] = Vector(
  ListPartiesResult(
    party = Bob::12209795b95e...,
    participants = Vector(
      ParticipantDomains(
        participant = PAR::participant2::12209795b95e...,
        domains = Vector(
          DomainPermission(domain = mydomain::1220ff0ca3d6..., permission =
↪Submission)
        )
      )
    )
  )
)
```

3.2.2.6 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")
res19: Map[com.digitalasset.canton.console.ParticipantReference, String] = Map(
  Participant 'participant1' ->
↪"1220d9e076507458dd31eafdb312c1f0f139997626b02d4afbfe3bf43d9e0f6b22ae",
  Participant 'participant2' ->
↪"1220d9e076507458dd31eafdb312c1f0f139997626b02d4afbfe3bf43d9e0f6b22ae"
)
```

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
res20: Seq[com.digitalasset.canton.console.LocalInstanceReference] =
↪ArraySeq(Participant 'participant1', Participant 'participant2', Domain
↪'mydomain')
```

or on the specific node type:

```
@ participants.all
res21: 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()
res22: Seq[com.digitalasset.canton.participant.admin.v0.DarDescription] = Vector(
  DarDescription(
    hash = "1220c5a4ac582223dcf2a59d323e474b3411df96f39cfa1304e2739ab7ca97f3b6b8",
    name = "AdminWorkflows"
  ),
  DarDescription(
    hash = "1220d9e076507458dd31eafdb312c1f0f139997626b02d4afbfe3bf43d9e0f6b22ae",
    name = "CantonExamples"
  )
)
```

and on the second participant, run:

```
@ participant2.dars.list()
res23: Seq[com.digitalasset.canton.participant.admin.v0.DarDescription] = Vector(
  DarDescription(
    hash = "1220c5a4ac582223dcf2a59d323e474b3411df96f39cfa1304e2739ab7ca97f3b6b8",
    name = "AdminWorkflows"
  ),
```

(continues on next page)

(continued from previous page)

```
  DarDescription(
    hash = "1220d9e076507458dd31eafdb312c1f0f139997626b02d4afbfe3bf43d9e0f6b22ae",
    name = "CantonExamples"
  )
)
```

One important observation is that you can not 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.

3.2.2.7 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::12209795b95e...
```

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
```

(continues on next page)

(continued from previous page)

```
    payer: Party
    owner: Party
    amount: Amount
    viewers: [Party]
  where

    ensure (amount.value >= 0.0)

    signatory payer
    observer viewers

    controller owner can
      Call : ContractId GetCash do
        create GetCash with payer; owner; amount
      Transfer : ContractId Iou
         with newOwner: Party do
           create this with owner = newOwner; viewers = []
      Share : ContractId Iou
            with viewer : Party
          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
    controller houseOwner can
      AcceptByOwner : ContractId Iou with iouId : ContractId Iou
        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 is using 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 = "105534e25027240ebf2c2554b200f679b70f53ae502c7eaf1f31448a7e283ea9",
  sourceDescription = "CantonExamples"
)
```

Using this package-id, we can create the paint offer:

```
@ 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 =
↪"105534e25027240ebf2c2554b200f679b70f53ae502c7eaf1f31448a7e283ea9",
..
```

and then send that command to the Ledger API:

```
@ participant2.ledger_api.commands.submit(Seq(bank), Seq(createIouCmd))
res27: com.daml.ledger.api.v1.transaction.TransactionTree = TransactionTree(
  transactionId =
↪"1220a9db068808be3a3a813a43cf36ee4321945e4add474ebc3b9b3892c3f9c32859",
  commandId = "83b0dc5c-0738-4d0e-8373-911333699a01",
  workflowId = "",
  effectiveAt = Some(
    value = Timestamp(
      seconds = 1646215864L,
      nanos = 948328000,
      unknownFields = UnknownFieldSet(fields = Map())
    )
  ),
  offset = "00000000000000000f",
..
```

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.
```

(continues on next page)

(continued from previous page)

```
  GrpcClientError: INVALID_ARGUMENT/DAML_AUTHORIZATION_ERROR(8,29399088):
↪Interpretation error: Error: node NodeId(0)
↪(105534e25027240ebf2c2554b200f679b70f53ae502c7eaf1f31448a7e283ea9:Iou:Iou)
↪requires authorizers
↪Bank::12209795b95ead3212d0d792d953860a30686cb78e04fa6af7e968db894c13fe6481, but
↪only
↪Alice::1220ef59f36b4a1a62b33eb463d67bfb5c7835151e8ef6e7b8774847dfb9daba5c4e
↪were given
  Request: SubmitAndWaitTransactionTree(actAs = Alice::1220ef59f36b..., commandId
↪= '', workflowId = '', submissionId = '', deduplicationPeriod = None(),
↪ledgerId = 'participant1', commands = ...)
  CorrelationId: 29399088-6623-4a94-8237-731ad943b4ea
..
```

And Alice can not 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,b973f0f3): This participant can not submit as the given submitter on
↪any connected domain
  Request: SubmitAndWaitTransactionTree(actAs = Bank::12209795b95e..., commandId
↪= '', workflowId = '', submissionId = '', deduplicationPeriod = None(),
↪ledgerId = 'participant1', commands = ...)
  CorrelationId: b973f0f308cd2812a778b4bd6bb26a5a
..
```

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 ==
↪"Iou.Iou")
aliceIou : com.digitalasset.canton.admin.api.client.commands.
↪LedgerApiTypeWrappers.WrappedCreatedEvent = WrappedCreatedEvent(
  event = CreatedEvent(
    eventId = "
↪#1220a9db068808be3a3a813a43cf36ee4321945e4add474ebc3b9b3892c3f9c32859:0",
    contractId =
↪"0014b14c38d58696831838107c6cfd163aa360033d487ad4d354258ab4662b7080ca001220b7cc36e360ee2e
↪",
..
```

We can check Alice's ACS, which will show us all the contracts Alice knows about:

```
@ participant1.ledger_api.acs.of_party(alice)
res29: Seq[com.digitalasset.canton.admin.api.client.commands.
↪LedgerApiTypeWrappers.WrappedCreatedEvent] = List(
  WrappedCreatedEvent(
    event = CreatedEvent(
      eventId = "
↪#1220a9db068808be3a3a813a43cf36ee4321945e4add474ebc3b9b3892c3f9c32859:0",
      contractId =
↪"0014b14c38d58696831838107c6cfd163aa360033d487ad4d354258ab4662b7080ca001220b7cc36e360ee2e
↪",
```

(continues on next page)

(continued from previous page)

```
      templateId = Some(
        value = Identifier(
          packageId =
↪"105534e25027240ebf2c2554b200f679b70f53ae502c7eaf1f31448a7e283ea9",
..
```

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))
res30: Seq[(String, Map[String, Any])] = List(
  (
    "Iou.Iou",
    HashMap(
      "payer" ->
↪"Bank::12209795b95ead3212d0d792d953860a30686cb78e04fa6af7e968db894c13fe6481",
      "viewers" -> List(elements = Vector()),
      "owner" ->
↪"Alice::1220ef59f36b4a1a62b33eb463d67bfb5c7835151e8ef6e7b8774847dfb9daba5c4e",
      "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 = "105534e25027240ebf2c2554b200f679b70f53ae502c7eaf1f31448a7e283ea9",
  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 in 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 =
↪"105534e25027240ebf2c2554b200f679b70f53ae502c7eaf1f31448a7e283ea9",
..
```

```
@ participant2.ledger_api.commands.submit_flat(Seq(bob), Seq(createOfferCmd))
res33: com.daml.ledger.api.v1.transaction.Transaction = Transaction(
  transactionId =
↪"1220effcce85b910a111bf6f7547a09b968264d707cb3a05344362424fbc35b096df",
  commandId = "913f46ae-2be1-42f1-bb31-ebd049061c3d",
  workflowId = "",
  effectiveAt = Some(
    value = Timestamp(
..
```

Alice will observe this offer on her node:

```
@ val paintOffer = participant1.ledger_api.acs.find_generic(alice, _.templateId
↪== "Paint.OfferToPaintHouseByPainter")
paintOffer : com.digitalasset.canton.admin.api.client.commands.
↪LedgerApiTypeWrappers.WrappedCreatedEvent = WrappedCreatedEvent(
  event = CreatedEvent(
    eventId = "
↪#1220effcce85b910a111bf6f7547a09b968264d707cb3a05344362424fbc35b096df:0",
    contractId =
↪"00137b4322a3fae11dba0386bb85ab7b5fbe34a0a9440205704026824e9d51af40ca001220e529d0cdafc826
↪",
    templateId = Some(
      value = Identifier(
..
```

3.2.2.8 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))
res35: Seq[(String, Map[String, Any])] = List(
  (
    "Paint.OfferToPaintHouseByPainter",
    HashMap(
      "painter" ->
↪"Bob::12209795b95ead3212d0d792d953860a30686cb78e04fa6af7e968db894c13fe6481",
      "houseOwner" ->
↪"Alice::1220ef59f36b4a1a62b33eb463d67bfb5c7835151e8ef6e7b8774847dfb9daba5c4e",
      "bank" ->
↪"Bank::12209795b95ead3212d0d792d953860a30686cb78e04fa6af7e968db894c13fe6481",
      "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))
res36: Seq[(String, Map[String, Any])] = List(
  (
    "Iou.Iou",
    HashMap(
      "payer" ->
↪"Bank::12209795b95ead3212d0d792d953860a30686cb78e04fa6af7e968db894c13fe6481",
```

(continues on next page)

(continued from previous page)

```
      "viewers" -> List(elements = Vector()),
      "owner" ->
↪"Alice::1220ef59f36b4a1a62b33eb463d67bfb5c7835151e8ef6e7b8774847dfb9daba5c4e",
      "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))
res37: Seq[(String, Map[String, Any])] = List(
  (
    "Iou.Iou",
    HashMap(
      "payer" ->
↪"Bank::12209795b95ead3212d0d792d953860a30686cb78e04fa6af7e968db894c13fe6481",
      "viewers" -> List(elements = Vector()),
      "owner" ->
↪"Alice::1220ef59f36b4a1a62b33eb463d67bfb5c7835151e8ef6e7b8774847dfb9daba5c4e",
      "amount.currency" -> "EUR",
      "amount.value" -> "100.0000000000"
    )
  ),
  (
    "Paint.OfferToPaintHouseByPainter",
    HashMap(
      "painter" ->
↪"Bob::12209795b95ead3212d0d792d953860a30686cb78e04fa6af7e968db894c13fe6481",
      "houseOwner" ->
↪"Alice::1220ef59f36b4a1a62b33eb463d67bfb5c7835151e8ef6e7b8774847dfb9daba5c4e",
      "bank" ->
↪"Bank::12209795b95ead3212d0d792d953860a30686cb78e04fa6af7e968db894c13fe6481",
      "amount.currency" -> "EUR",
      "amount.value" -> "100.0000000000"
    )
  )
)
```

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 =
↪"105534e25027240ebf2c2554b200f679b70f53ae502c7eaf1f31448a7e283ea9",
..
```

```
@ participant1.ledger_api.commands.submit_flat(Seq(alice), Seq(acceptOffer))
res40: com.daml.ledger.api.v1.transaction.Transaction = Transaction(
  transactionId =
↪"122033796b6359dd35f5d7c918819a8567d82d7873a1af1b7c598d9bf0975331b2be",
  commandId = "4d78921d-9e9d-42fd-894b-5b31207015fe",
  workflowId = "",
  effectiveAt = Some(
    value = Timestamp(
..
```

Note that the conversion to `LfContractId` was required to pass in the Iou contract id as the correct type.

3.2.2.9 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.

3.2.2.10 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
```

(continues on next page)

(continued from previous page)

```
// 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)

// 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", "http://localhost:5018")

// 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 {
    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.

3.2.2.11 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*.

3.2.3 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.

3.2.3.1 Starting Canton

Follow the Daml SDK installation guide to get the SDK locally installed.

This guide has been tested with the SDK version 2.0.0. Set the environment variable `DAML_SDK_VERSION` to `2.0.0` so that subsequent `daml` commands use this version.

```
export DAML_SDK_VERSION=2.0.0
```

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.

3.2.3.2 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=2.0.0 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=2.0.0 daml json-api \
   --ledger-host localhost \
   --ledger-port 12021 \
   --http-port 7576 \
   --allow-insecure-tokens
```

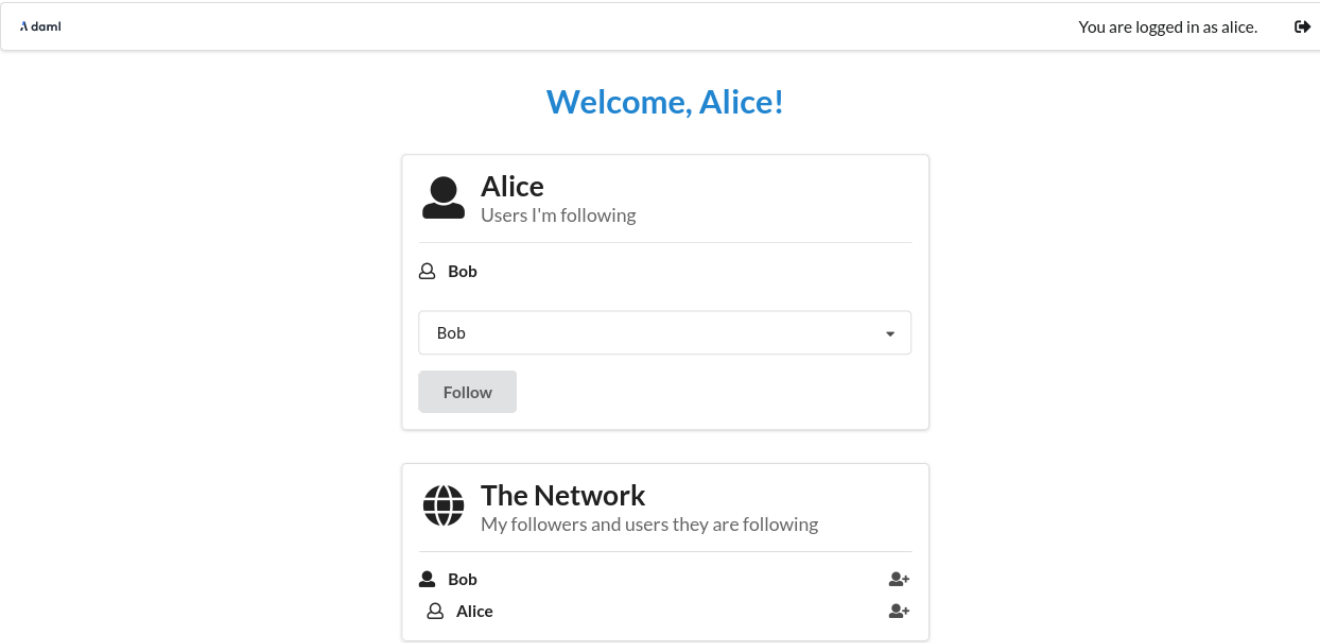
Then start another instance of the UI for Bob, running on port `3001` and connected to the HTTP JSON API on port `7576`:

```
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.

3.2.3.3 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.

3.2.4 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 can not 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.

3.2.4.1 Part 1: A multi-domain workflow

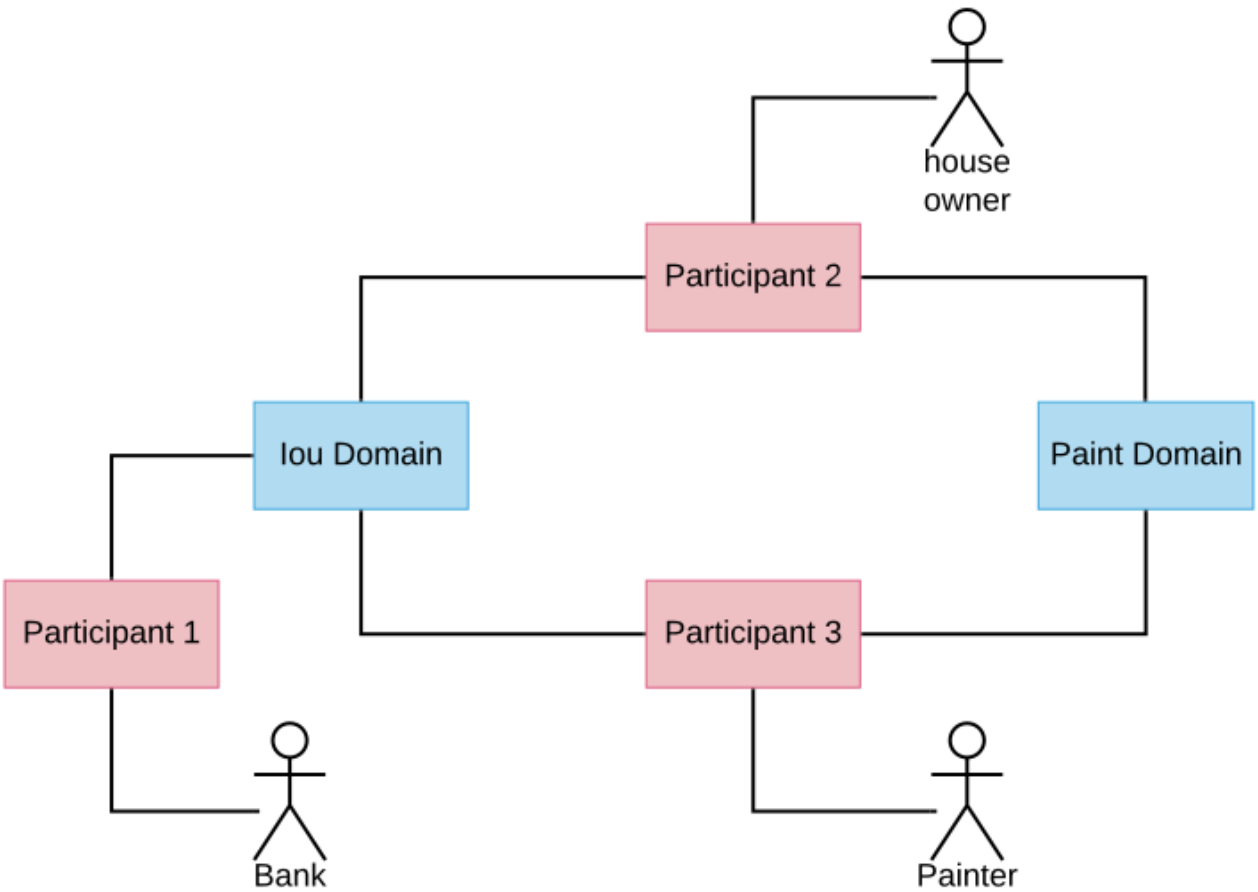
We consider the *paint agreement scenario* from the *Getting started* tutorial. The house owner and the painter want to enter a paint agreement that obliges the painter to paint the house owner's house. To enter such an agreement, the house owner proposes a paint offer to the painter and the painter 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.

```
canton {
  features {
    enable-preview-commands = yes
    enable-testing-commands = yes
  }
  monitoring {
    tracing.propagation = enabled
    log-message-payloads = true
  }
  domains {
    iou {
      public-api.port = 13018
      admin-api.port = 13019
      storage.type = memory
      domain-parameters.unique-contract-keys = false
    }

    paint {
      public-api.port = 13028
      admin-api.port = 13029
      storage.type = memory
      domain-parameters.unique-contract-keys = false
    }
  }

  participants {
    participant1 {
      ledger-api.port = 13011
      admin-api.port = 13012
      storage.type = memory
      parameters.unique-contract-keys = false
    }

    participant2 {
      ledger-api.port = 13021
      admin-api.port = 13022
      storage.type = memory
      parameters.unique-contract-keys = false
    }

    participant3 {
      ledger-api.port = 13031
      admin-api.port = 13032
      storage.type = memory
      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_parameters(
  _.copy(transferExclusivityTimeout = TimeoutDuration.Zero)
) // disables automatic transfer-in

paint.service.update_dynamic_parameters(
  _.copy(transferExclusivityTimeout = TimeoutDuration.ofSeconds(2))
)

// 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(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.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))
```

Transferring a contract

In Canton, contracts reside on at most one domain at a time. 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,                     // Origin domain
  iouAlias                        // Target domain
)
```

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)
```

(continues on next page)

(continued from previous page)

```
// 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

Contracts reside on domains.
Stakeholders can move contracts from one domain to another using `transfer.execute`. All stakeholders must be connected to the origin 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.

3.2.4.2 Part 2: Composing existing workflows

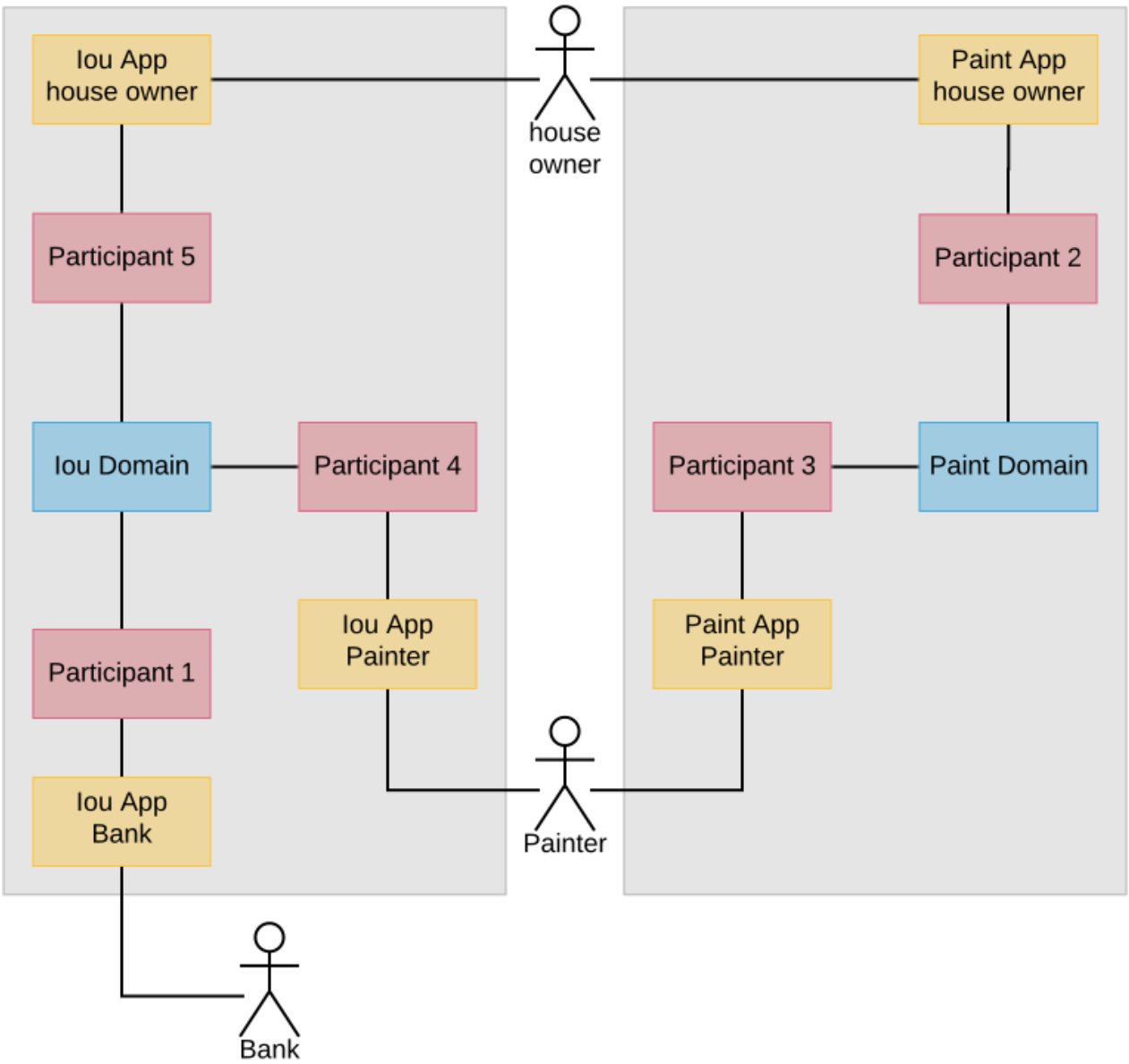
This part shows how existing workflows can be composed even if they work on separate domains. The running example is a variation of the paint example from the first part with a more complicated topology. We therefore assume that you have gone through *the first part* of this tutorial. Technically, this 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.

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

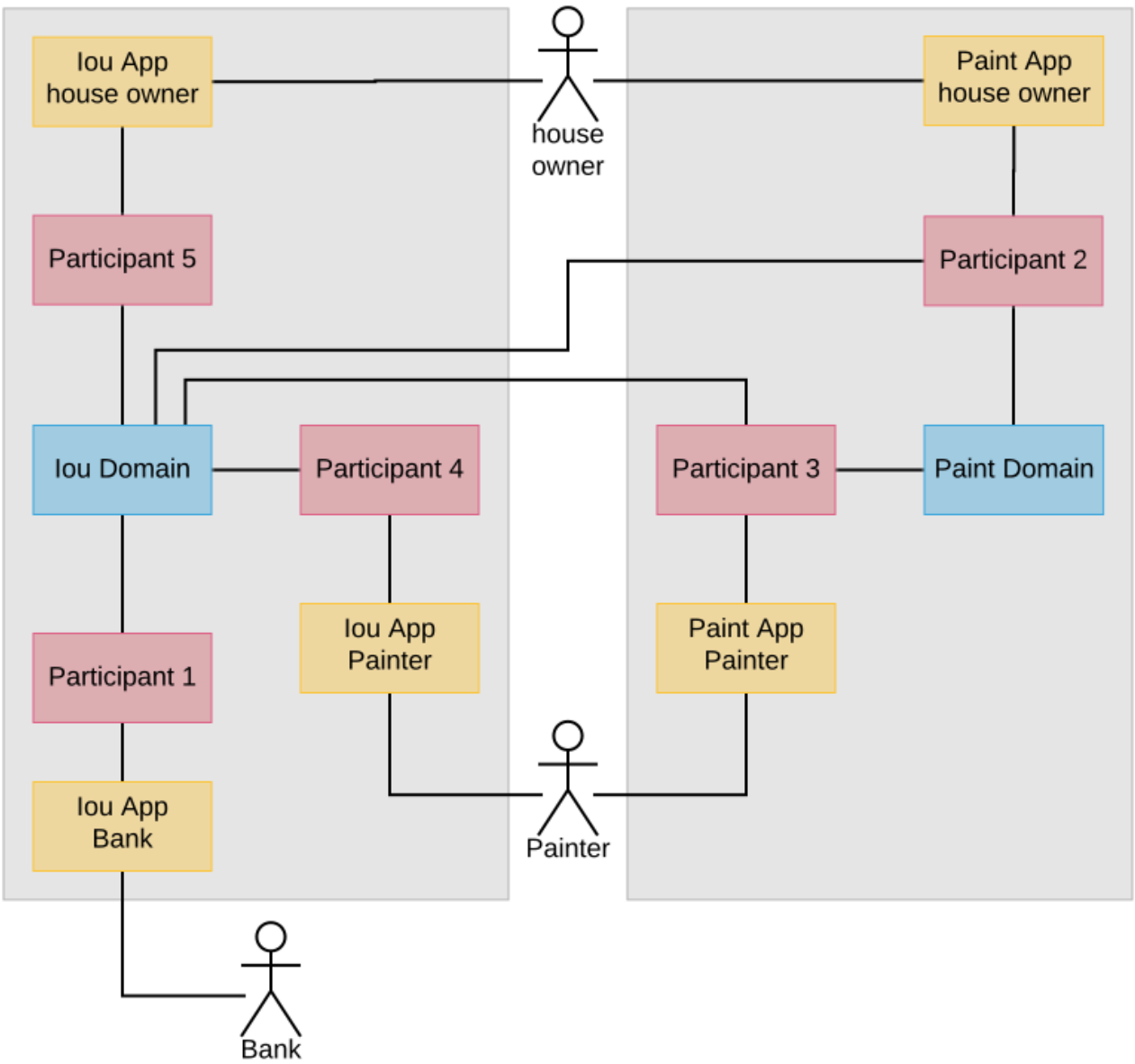
We now show how the house owner and the painter can avoid need for out-of-band coordination when entering in paint agreements. The goal is to reuse the existing infrastructure for managing IOUs and paint agreements as much as possible. The following changes are needed:

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
      parameters.unique-contract-keys = false
    }

    participant5 {
      ledger-api.port = 13051
      admin-api.port = 13052
      storage.type = memory
      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.


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

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_parameters(
  _.copy(transferExclusivityTimeout = TimeoutDuration.Zero)
) // disables automatic transfer-in

paint.service.update_dynamic_parameters(
  _.copy(transferExclusivityTimeout = TimeoutDuration.ofSeconds(2))
)

// 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)

// enable the house owner on participant 5 and the painter on participant 4
// as explained in the identity management documentation at
// https://www.canton.io/docs/stable/user-manual/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)
```

(continues on next page)

(continued from previous page)

```
// 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.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
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
```

(continues on next page)

(continued from previous page)

```
).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 origin domain, in this case the `paint` domain. Any stakeholder whose participant is connected to the origin 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,                     // Origin 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.

3.3 User Manual

3.3.1 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.

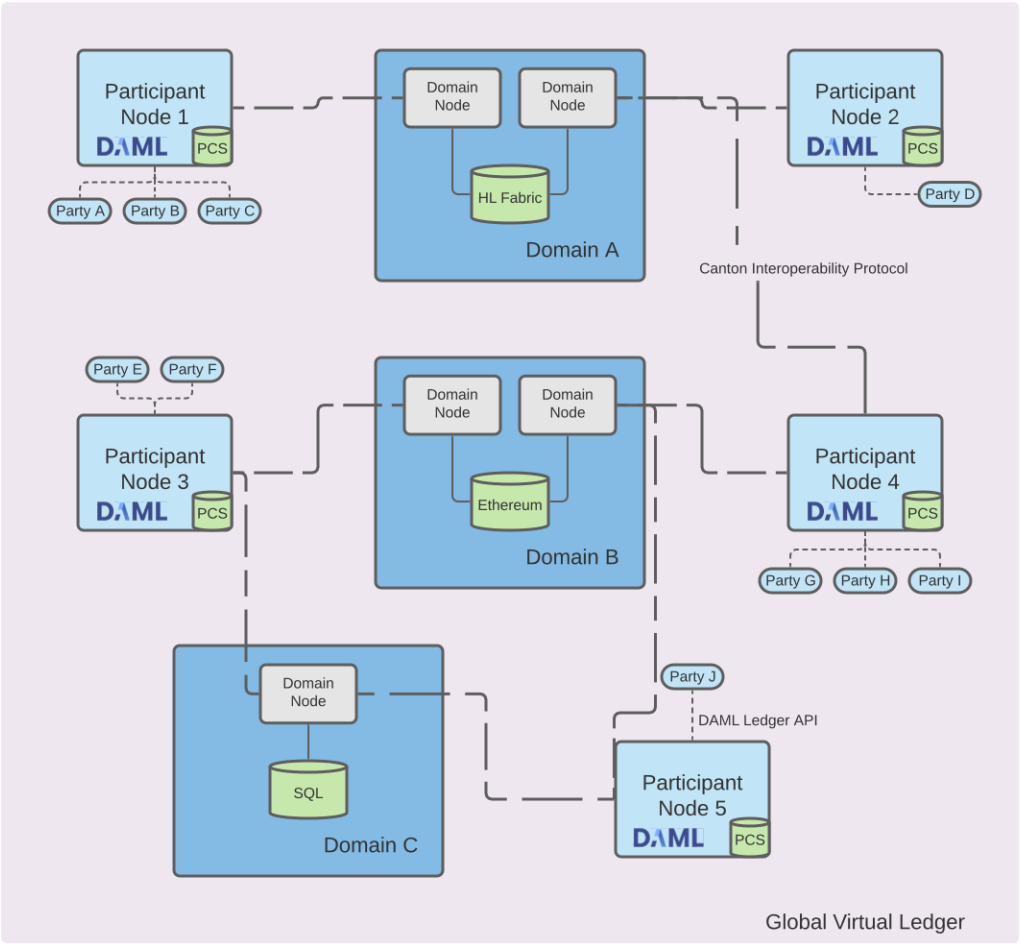
3.3.1.1 Downloading Canton

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.

3.3.1.2 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, 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.

3.3.1.3 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"
```

3.3.1.4 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:11
docker run --rm --name pg-docker -e POSTGRES_PASSWORD=docker -d -p 5432:5432
↪postgres:11
```

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 ...
```

3.3.1.5 Setting up a Participant

Now that you have made your persistence choice (assuming Postgres here), 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 configuration 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 the administration API or the console.

Note: Please consult the section on the *console commands* and *administration APIs*.

If you don't know how to connect to domains, onboard parties or provision Daml code, please read the *getting started guide*.

3.3.1.6 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. In-Process Postgres based domain (simplest choice)
2. Hyperledger Fabric based domain
3. Secure enclave based domain
4. Ethereum based domain (demo)

This manual will explain you how to setup an in-process based domain using Postgres. All other domains are a set of microservices and part of the Enterprise edition. However, 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, Ethereum or enclave 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/domain.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.

3.3.1.7 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 $CONF/nodes/domain.conf,
↪$CONF/nodes/participant1.conf
```

3.3.2 Running in Docker

3.3.2.1 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 Enterprise edition, you can download it using

```
docker login digitalasset-canton-enterprise-docker.jfrog.io
docker pull digitalasset-canton-enterprise-docker.jfrog.io/digitalasset/canton-
↪enterprise
```

3.3.2.2 Starting Canton

The canton executable is the default image entry point so all examples using `bin/canton` can simply substitute that with `docker run digtialasset/canton`.

For example, to run our example simple topology:

```
docker run --rm -it digitalasset/canton-open-source:latest --config examples/01-
↪simple-topology/simple-topology.conf --bootstrap examples/01-simple-topology/
↪simple-ping.canton
```

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`. This directory contains the same content as the release archive (`daml`, `dar`, `examples`).

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:2.0.0`.

3.3.2.3 Configuring Logging

The default convention with logging of containers is to have the process to log to `stdout`. Therefore, we to change the logging behaviour of Canton using appropriate *command line flags*, such as `--log-profile=container`.

3.3.2.4 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.

3.3.2.5 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 \
   -Dcanton.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.

3.3.2.6 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:11 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.

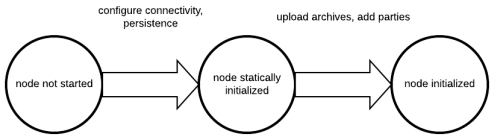
Note that you also need to create the databases yourself, which for Postgres you can do using `psql`

```
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.

3.3.3 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 the *administration APIs* or the *console*.



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
```

(continues on next page)

(continued from previous page)

```
      admin-api.port = 5019
    }
  }
  // enable ledger_api commands for our getting started guide
  features.enable-testing-commands = yes
}
```

3.3.3.1 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.

3.3.3.2 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.

3.3.3.3 Advanced Configurations

Configuration files can be nested and combined together. First, using the `include` directive (with relative paths), a configuration file can include other configuration files.

```
canton {
    domains {
        include "domain1.conf"
    }
}
```

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.

3.3.3.4 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 {
  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}
      }
    }
    // 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.
    max-connections = ${?POSTGRES_NUM_CONNECTIONS}
  }
}
```

Such a definition can subsequently be referenced in the actual node definition:

```
canton {
    domains {
        mydomain {
```

(continues on next page)

(continued from previous page)

```
            storage = ${_shared.storage}
            storage.config.properties.databaseName = ${CANTON_DB_NAME_DOMAIN}
        }
    }
}
```

3.3.3.5 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
            domain-parameters.unique-contract-keys = false
        }
    }
    participants {
        participant1 {
            // subsequent changes have no effect and the mode of a node can never
↪be changed
            parameters.unique-contract-keys = false
        }
    }
}
```

Please note that the setting is final and can not be changed subsequently. We will provide a migration path once multi-domain is fully implemented.

3.3.3.6 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.init.startup-fail-fast = "no"
```

This option should be used with care as, by design, it can cause infinite, noisy waits.

3.3.3.7 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.

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.

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 {
```

(continues on next page)

(continued from previous page)

```
      dataSourceClass = "org.postgresql.ds.PGSimpleDataSource"
      properties = {
        serverName = "localhost"
        serverName = ${?POSTGRES_HOST}
        portNumber = "5432"
        portNumber = ${?POSTGRES_PORT}
        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.
    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.

Max Connection Settings

The storage configuration can further be tuned using the following additional setting:

```
canton.participants.<service-name>.storage.max-connections = X
```

This allows for setting the maximum number of DB connections used by a Canton node. If 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.

Additionally, the number of parallel indexer connections can be configured via

```
canton.participants.<service-name>.parameters.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.

3.3.3.8 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. All APIs are based on GRPC, which is an efficient RPC and streaming protocol with client support in almost all relevant programming languages. Native bindings can be built using the *API definitions*.

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) */
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)

/** Increase the default port number for each new instance by portStep */
private val portStep = 10
```

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.

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.

A full configuration example is given by

```
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.
    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=730

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
```

(continues on next page)

(continued from previous page)

```
  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"
#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"
```

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:

```
admin-api     {
  address = "127.0.0.1"
  port = 5022
  keep-alive-server {
    time = 40s
    timeout = 20s
    permit-keep-alive-time = 20s
  }
}
```

(continues on next page)

(continued from previous page)

```
sequencer-client {
  keep-alive-client {
    time = 60s
    timeout = 30s
  }
}
```

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.

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:

```
ledger-api {
  address = "127.0.0.1"
  port = 5021
  max-inbound-message-size = 20971520
}
```

3.3.3.9 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. Please consult the Daml SDK manual to understand the various configuration options and their security aspects. Canton exposes precisely the same JWT authorization options as explained therein.

In order to enable JWT authorization checks, your safe configuration options are

```
_shared {
  ledger-api {
    auth-services = [{
```

(continues on next page)

(continued from previous page)

```
      // 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)
      certificate = ${JWT_CERTIFICATE_FILE}
    }]
  }
}
```

```
_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}
    }]
  }
}
```

while there is also unsafe `HMAC256` based support, which can be enabled using

```
_shared {
  ledger-api {
    auth-services = [{
      type = unsafe-jwt-hmac-256
      secret = "not-safe-for-production"
    }]
  }
}
```

Note that you can define several authorization plugins. If several are defined, the system will use the claim of the first auth plugin that does not return Unauthorized.

3.3.3.10 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 will connect to. There is a built in authentication of the restricted services on the public 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.

As with the admin-api, network traffic can (and should be) encrypted using TLS.

An example configuration section which enables TLS encryption and server-side TLS authentication is given by

```
public-api {
  port = 5028
  address = localhost // defaults to 127.0.0.1
```

(continues on next page)

(continued from previous page)

```
  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",
  certificatesPath = certs, // path to certificate chain file (.pem) of server
)
```

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:

```
domain-parameters {
  // example setting
  unique-contract-keys = yes
}
```

The full set of available parameters can be found in the scala reference documentation.

3.3.3.11 Limiting concurrent GRPC requests (preview feature)

In large-scale deployments a Canton node may get more GRPC requests than it can cope with, leading to requests timing out. Canton has an experimental integration with concurrency-limits to limit the number of concurrent requests and prevent nodes from becoming overloaded.

Canton's GRPC services can be configured to use various options provided by the concurrency-limits library:

A fixed limit on concurrent requests

```
canton.participants.participant1.admin-api.concurrency-limit-beware-preview-
↪feature {
    type = "fixed-limit"
    limit = "10"
}
```

A dynamic limit for the number of concurrent requests, inspired by TCP Vegas

```
canton.participants.participant1.admin-api.concurrency-limit-beware-preview-
↪feature {
    # Values are defaults from https://github.com/Netflix/concurrency-limits
    type = "vegas-limit"
    max-concurrency = 1000
    smoothing = 1.0
}
```

A gradient-based algorithm to dynamically infer the concurrency limit

```
canton.participants.participant1.admin-api.concurrency-limit-beware-preview-
↪feature {
    # Values are defaults from https://github.com/Netflix/concurrency-limits
    type = "gradient-2-limit"
    max-concurrency = 200
    smoothing = 0.2
    rtt-tolerance = 1.5
}
```

Any of these options, with an added smoothing window

```
canton.participants.participant1.admin-api.concurrency-limit-beware-preview-
↪feature {
    # Values are defaults from https://github.com/Netflix/concurrency-limits
    type = "windowed-limit"
    window-size = 10
    delegate = {
        type = gradient-2-limit
        max-concurrency = 200
        smoothing = 0.2
        rtt-tolerance = 1.5
    }
}
```

See the concurrency-limits library for more information on these different options.

3.3.4 Canton Administration APIs

Canton provides a *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 of both types of nodes, participant and domain.

There are three categories of admin-apis: participant, domain and identity.

3.3.4.1 Participant Admin APIs

The participant exposes the following admin-api services:

Package Service

The package service is used to manage the installed packages.

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

syntax = "proto3";

package com.digitalasset.canton.participant.admin.v0;

import "google/protobuf/empty.proto";

service PackageService {

    // List all Daml-LF archives on the participant node - return their hashes
    rpc ListPackages(ListPackagesRequest) returns (ListPackagesResponse);

    // Lists all the modules in package on the participant node
    rpc ListPackageContents (ListPackageContentsRequest) returns
↪(ListPackageContentsResponse);

    // List all DARs on the participant node - return their hashes and filenames
    rpc ListDars(ListDarsRequest) returns (ListDarsResponse);

    // Upload a DAR file and all packages inside to the participant node
    rpc UploadDar (UploadDarRequest) returns (UploadDarResponse);

    rpc RemovePackage (RemovePackageRequest) returns (RemovePackageResponse);

    // Obtain a DAR file by hash -- for inspection & download
    rpc GetDar(GetDarRequest) returns (GetDarResponse);

    // Share a DAR with another participant
    rpc Share(ShareRequest) returns (google.protobuf.Empty);

    // List requests this participant has made to share DARs with another
↪participant
    rpc ListShareRequests(google.protobuf.Empty) returns
↪(ListShareRequestsResponse);

    // List offers to share a DAR that this participant has received
    rpc ListShareOffers(google.protobuf.Empty) returns (ListShareOffersResponse);

    // Accept a DAR sharing offer (this will install the DAR into the participant)
    rpc AcceptShareOffer(AcceptShareOfferRequest) returns (google.protobuf.Empty);

    // Reject a DAR sharing offer
    rpc RejectShareOffer(RejectShareOfferRequest) returns (google.protobuf.Empty);
```

(continues on next page)

(continued from previous page)

```
    // Add party to our DAR distribution whitelist
    rpc WhitelistAdd(WhitelistChangeRequest) returns (google.protobuf.Empty);

    // Remove party from our DAR distribution whitelist
    rpc WhitelistRemove(WhitelistChangeRequest) returns (google.protobuf.Empty);

    // List all parties currently on the whitelist
    rpc WhitelistList(google.protobuf.Empty) returns (WhitelistListResponse);
}

message ListPackageContentsRequest {
    string package_id = 1;
}

message ListPackageContentsResponse {
    repeated ModuleDescription modules = 1;
}

message RemovePackageRequest {
    string package_id = 1;
    bool force = 2;
}

message RemovePackageResponse {
    google.protobuf.Empty success = 1;
}

message ModuleDescription {
    string name = 1;
}

message ListPackagesRequest {
    int32 limit = 1;
}

message ListPackagesResponse {
    repeated PackageDescription package_descriptions = 1;
}

message ListDarsRequest {
    int32 limit = 1;
}

message ListDarsResponse {
    repeated DarDescription dars = 1;
}

message DarDescription {
    string hash = 1;
    string name = 2;
}

message UploadDarRequest {
    bytes data = 1;
    string filename = 2;
    // if set to true, we'll register the vetting topology transactions with the
↪idm
```

(continues on next page)

(continued from previous page)

```
    bool vet_all_packages = 3;
    // if set to true, we'll wait until the vetting transaction has been observed
↪by this participant on all connected domains
    bool synchronize_vetting = 4;
}

message UploadDarResponse {
    oneof value {
        Success success = 1;
        Failure failure = 2;
    }
    message Success {
        string hash = 1;
    }
    message Failure {
        string reason = 1;
    }
}

message GetDarRequest {
    string hash = 1;
}

message GetDarResponse {
    bytes data = 1;
    string name = 2;
}

message PackageDescription {
    string package_id = 1;
    string source_description = 3;
}

message ShareRequest {
    string dar_hash = 1;
    string recipient_id = 2;
}

message ListShareRequestsResponse {
    repeated Item share_requests = 1;

    message Item {
        string id = 1;
        string dar_hash = 2;
        string recipient_id = 3;
        string name = 4;
    }
}

message ListShareOffersResponse {
    repeated Item share_offers = 1;

    message Item {
        string id = 1;
        string dar_hash = 2;
        string owner_id = 3;
```

(continues on next page)

(continued from previous page)

```
        string name = 4;
    }
}

message AcceptShareOfferRequest {
    string id = 1;
}

message RejectShareOfferRequest {
    string id = 1;
    // informational message explaining why we decided to reject the DAR
    // can be empty
    string reason = 2;
}

message WhitelistChangeRequest {
    string party_id = 1;
}

message WhitelistListResponse {
    repeated string party_ids = 1;
}
```

Participant Status Service

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

syntax = "proto3";

package com.digitalasset.canton.health.admin.v0;
import "google/protobuf/empty.proto";
import "google/protobuf/duration.proto";
import "google/protobuf/wrappers.proto";

service StatusService {
    rpc Status(google.protobuf.Empty) returns (NodeStatus);
}

message TopologyQueueStatus {
    // how many topology changes are currently queued at the manager
    uint32 manager = 1;
    // how many topology changes are currently queued at the dispatcher
    uint32 dispatcher = 2;
    // how many topology changes are currently waiting to become effective
↪(across all connected domains in the case of participants)
    uint32 clients = 3;
}

message NodeStatus {
    message Status {
        string id = 1;
```

(continues on next page)

(continued from previous page)

```
        google.protobuf.Duration uptime = 2;
        map<string, int32> ports = 3;
        bytes extra = 4; // contains extra information depending on the node type
        bool active = 5; // Indicate if the node is active, usually true unless it
↪'s a replicated node that is passive
        TopologyQueueStatus topology_queues = 6; // indicates the state of the
↪topology queues (manager / dispatcher only where they exist)
    }

    message NotInitialized {
        bool active = 1; // Indicate if the node is active, usually true unless it
↪'s a replicated node that is passive
    }

    oneof response {
        NotInitialized not_initialized = 1; // node is running but has not been
↪initialized yet
        Status success = 2; // successful response from a running and initialized
↪node
    }
}



// domain node specific extra status info
message DomainStatusInfo {
    repeated string connected_participants = 1;
    // optional - only set if a sequencer is being run by the domain
    SequencerHealthStatus sequencer = 2;
}

// participant node specific extra status info
message ParticipantStatusInfo {
    message ConnectedDomain {
        string domain = 1;
        bool healthy = 2;
    }
    repeated ConnectedDomain connected_domains = 1;
    // Indicate if the participant node is active
    // True if the participant node is replicated and is the active replica, or
↪true if not replicated
    bool active = 2;
}

message SequencerNodeStatus {
    repeated string connected_participants = 1;
    // required - status of the sequencer component it is running
    SequencerHealthStatus sequencer = 2;
}

// status of the sequencer component
message SequencerHealthStatus {
    // is the sequencer component active - can vary by implementation for what
↪this means
    bool active = 1;
    // optionally set details on how sequencer is healthy/unhealthy
```

(continues on next page)

(continued from previous page)

```
    google.protobuf.StringValue details = 2;
}
```

Ping Pong Service

Canton uses a default simple ping-pong workflow to smoke-test a deployment.

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

syntax = "proto3";

package com.digitalasset.canton.participant.admin.v0;

service PingService {
    rpc ping(PingRequest) returns (PingResponse);
}

message PingRequest {

    repeated string target_parties = 1;
    repeated string validators = 2;
    uint64 timeout_milliseconds = 3;
    uint64 levels = 4;
    uint64 grace_period_milliseconds = 5;
    string workflow_id = 6; // optional
    string id = 7; // optional UUID to be used for ping test

}

message PingSuccess {
    uint64 ping_time = 1;
    string responder = 2;
}

message PingFailure {
}

message PingResponse {
    oneof response {
        PingSuccess success = 1;
        PingFailure failure = 2;
    }

}
```

Domain Connectivity Service

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

syntax = "proto3";

package com.digitalasset.canton.participant.admin.v0;

import "google/protobuf/duration.proto";
import "com/digitalasset/canton/time/admin/v0/time_tracker_config.proto";
import "com/digitalasset/canton/protocol/v0/sequencing.proto";

/**
 * Domain connectivity service for adding and connecting to domains
 *
 * The domain connectivity service allows to register to new domains and control
↪the
 * participants domain connections.
 */
service DomainConnectivityService {
    // reconnect to domains
    rpc ReconnectDomains(ReconnectDomainsRequest) returns
↪(ReconnectDomainsResponse);
    // configure a new domain connection
    rpc RegisterDomain(RegisterDomainRequest) returns (RegisterDomainResponse);
    // reconfigure a domain connection
    rpc ModifyDomain(ModifyDomainRequest) returns (ModifyDomainResponse);
    // connect to a configured domain
    rpc ConnectDomain(ConnectDomainRequest) returns (ConnectDomainResponse);
    // disconnect from a configured domain
    rpc DisconnectDomain(DisconnectDomainRequest) returns
↪(DisconnectDomainResponse);
    // list connected domains
    rpc ListConnectedDomains(ListConnectedDomainsRequest) returns
↪(ListConnectedDomainsResponse);
    // list configured domains
    rpc ListConfiguredDomains(ListConfiguredDomainsRequest) returns
↪(ListConfiguredDomainsResponse);
    // Get the service agreement for the domain
    rpc GetAgreement(GetAgreementRequest) returns (GetAgreementResponse);
    // Accept the agreement of the domain
    rpc AcceptAgreement(AcceptAgreementRequest) returns (AcceptAgreementResponse);
    // Get the domain id of the given domain alias
    rpc GetDomainId(GetDomainIdRequest) returns (GetDomainIdResponse);
}

message DomainConnectionConfig {
    // participant local identifier of the target domain
    string domain_alias = 1;
    // connection information to sequencer
    com.digitalasset.canton.protocol.v0.SequencerConnection sequencerConnection =
↪2;
    // if false, then domain needs to be manually connected to (default false)
    bool manual_connect = 3;
```

(continues on next page)

(continued from previous page)

```
    // optional domainId (if TLS isn't to be trusted)
    string domain_id = 4;
    // optional. Influences whether the participant submits to this domain, if
↪several domains are eligible
    int32 priority = 5;
    // initial delay before an attempt to reconnect to the sequencer
    google.protobuf.Duration initialRetryDelay = 6;
    // maximum delay before an attempt to reconnect to the sequencer
    google.protobuf.Duration maxRetryDelay = 7;
    // configuration for how time is tracked and requested on this domain
    com.digitalasset.canton.time.admin.v0.DomainTimeTrackerConfig timeTracker = 8;
}

message ReconnectDomainsRequest {
    /* if set to true, the connection attempt will succeed even if one of the
↪connection attempts failed */
    bool ignore_failures = 1;

}
message ReconnectDomainsResponse {

}

/** Register and optionally auto-connect to a new domain */
message RegisterDomainRequest {
    DomainConnectionConfig add = 1;
}

message RegisterDomainResponse {

}

message ModifyDomainRequest {
    DomainConnectionConfig modify = 1;
}

message ModifyDomainResponse {

}

message ListConfiguredDomainsRequest {

}

message ListConfiguredDomainsResponse {
    message Result {
        DomainConnectionConfig config = 1;
        bool connected = 2;
    }
    repeated Result results = 1;
}

message ConnectDomainRequest {
    string domain_alias = 1;
    /* if retry is set to true, we will keep on retrying if the domain is
↪unavailable */
```

(continues on next page)

(continued from previous page)

```
    bool retry = 2;
}

message ConnectDomainResponse {
    /* true if the domain is connected, false if the domain is offline, exception
↪on any other error */
    bool connected_successfully = 1;
}

message DisconnectDomainRequest {
    string domain_alias = 1;
}

message DisconnectDomainResponse {
}

message ListConnectedDomainsRequest {
}

message ListConnectedDomainsResponse {
    message Result {
        string domain_alias = 1;
        string domain_id = 2;
        bool healthy = 3;
    }
    repeated Result connected_domains = 1;
}

message GetAgreementRequest {
    string domain_alias = 1;
}

message GetAgreementResponse {
    string domain_id = 1;
    Agreement agreement = 2;
    bool accepted = 3;
}

message Agreement {
    string id = 1;
    string text = 2;
}

message AcceptAgreementRequest {
    string domain_alias = 1;
    string agreement_id = 2;
}

message AcceptAgreementResponse {
}

message GetDomainIdRequest {
    string domain_alias = 1;
}

message GetDomainIdResponse {
```

(continues on next page)

(continued from previous page)

```
    string domain_id = 2;
}
```

Party Name Management Service

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

syntax = "proto3";

package com.digitalasset.canton.participant.admin.v0;

/**
  * Local participant service allowing to set the display name for a party
  *
  * The display name is a local property to the participant. The participant is
↪encouraged to perform
  * a Daml based KYC process and add some automation which will update the
↪display names based
  * on the desired update rules.
  *
  * As such, this function here just offers the bare functionality to perform
↪this.
  */
service PartyNameManagementService {
    rpc setPartyDisplayName(SetPartyDisplayNameRequest) returns
↪(SetPartyDisplayNameResponse);
}

message SetPartyDisplayNameRequest {
    string party_id = 1;
    string display_name = 2;
}

message SetPartyDisplayNameResponse {

}
```

Inspection Service

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

syntax = "proto3";

package com.digitalasset.canton.participant.admin.v0;

import "google/protobuf/timestamp.proto";
```

(continues on next page)

(continued from previous page)

```
/**
 * Inspection Service
 *
 * Supports inspecting the Participant for details about its ledger.
 * This contains only a subset of the ParticipantInspection commands that can run
 ↪over the admin-api instead of requiring
 * direct access to the participant node instance.
 */
service InspectionService {
    // Lookup the domain where a contract is currently active.
    // Supports querying many contracts at once.
    rpc LookupContractDomain(LookupContractDomain.Request) returns
 ↪(LookupContractDomain.Response);
    // Lookup the domain that the transaction was committed over. Can fail with
 ↪NOT_FOUND if no domain was found.
    rpc LookupTransactionDomain(LookupTransactionDomain.Request) returns
 ↪(LookupTransactionDomain.Response);
    // Look up the ledger offset corresponding to the timestamp, specifically the
 ↪largest offset such that no later
    // offset corresponds to a later timestamp than the specified one.
    rpc LookupOffsetByTime(LookupOffsetByTime.Request) returns
 ↪(LookupOffsetByTime.Response);
    // Look up the ledger offset by an index, e.g. 1 returns the first offset, 2
 ↪the second, etc.
    rpc LookupOffsetByIndex(LookupOffsetByIndex.Request) returns
 ↪(LookupOffsetByIndex.Response);
}

message LookupContractDomain {
    message Request {
        // set of contract ids to lookup their active domain aliases.
        repeated string contract_id = 1;
    }

    message Response {
        // map of contract id to domain alias.
        // if a request contract id from the request is missing from this map it
 ↪indicates that the contract was not
        // active on any current domain.
        map<string, string> results = 1;
    }
}

message LookupTransactionDomain {
    message Request {
        // the transaction to look up
        string transaction_id = 1;
    }

    message Response {
        // the domain that the transaction was committed over
        string domain_id = 1;
    }
}

message LookupOffsetByTime {
```

(continues on next page)

(continued from previous page)

```
    message Request {
        // the timestamp to look up the offset for
        google.protobuf.Timestamp timestamp = 1;
    }

    message Response {
        // the absolute offset as a string corresponding to the specified
↪timestamp.
        // empty string if no such offset exists.
        string offset = 1;
    }
}

message LookupOffsetByIndex {
    message Request {
        // the index to look up the offset for, needs to be 1 or larger
        int64 index = 1;
    }

    message Response {
        // the absolute offset as a string corresponding to the specified index.
        // empty string if no such offset exists.
        string offset = 1;
    }
}
```

Transfer Service

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

syntax = "proto3";

package com.digitalasset.canton.participant.admin.v0;

import "google/protobuf/timestamp.proto";
import "com/digitalasset/canton/protocol/v0/synchronization.proto";

// Supports transferring contracts from one domain to another
service TransferService {
    // transfer out a contract
    rpc TransferOut (AdminTransferOutRequest) returns (AdminTransferOutResponse);

    // transfer-in a contract
    rpc TransferIn (AdminTransferInRequest) returns (AdminTransferInResponse);

    // return the in-flight transfers on a given participant for a given target
↪domain
    rpc TransferSearch (AdminTransferSearchQuery) returns
↪(AdminTransferSearchResponse);
}
```

(continues on next page)

(continued from previous page)

```
message AdminTransferOutRequest {
    string submitting_party = 1;
    string contract_id = 2;
    string origin_domain = 3;
    string target_domain = 4;
}

message AdminTransferOutResponse {
    com.digitalasset.canton.protocol.v0.TransferId transfer_id = 1;
}

message AdminTransferInRequest {
    string submitting_party_id = 1;
    string target_domain = 2;
    com.digitalasset.canton.protocol.v0.TransferId transfer_id = 3;
}

message AdminTransferInResponse {

}

message AdminTransferSearchQuery {
    string search_domain = 1;
    string filter_origin_domain = 2; // exact match if non-empty
    google.protobuf.Timestamp filter_timestamp = 3; // optional; exact match if
↪set
    string filter_submitting_party = 4;
    int64 limit = 5;
}

message AdminTransferSearchResponse {
    repeated TransferSearchResult results = 1;

    message TransferSearchResult {
        string contract_id = 1;
        com.digitalasset.canton.protocol.v0.TransferId transfer_id = 2;
        string origin_domain = 3;
        string target_domain = 4;
        string submitting_party = 5;
        bool ready_for_transfer_in = 6;
    }
}
```

Pruning Service

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

syntax = "proto3";

package com.digitalasset.canton.participant.admin.v0;
```

(continues on next page)

(continued from previous page)

```
// Canton-internal pruning service that prunes only canton state, but leaves the
↪ledger-api
// state unpruned.
service PruningService {

    // Prune the participant 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 one of the following errors:
    // - ``INVALID_ARGUMENT``: if the payload, particularly the offset is
↪malformed or missing
    // - ``INTERNAL``: if the participant has encountered a failure and has
↪potentially
    //   applied pruning partially. Such cases warrant verifying the participant
↪health before
    //   retrying the prune with the same (or a larger, valid) offset. Successful
↪retries
    //   after such errors ensure that different components reach a consistent
↪pruning state.
    // - ``FAILED_PRECONDITION``: if the participant is not yet able to prune at
↪the specified
    //   offset or if pruning is invoked on a participant running the Community
↪Edition.
    rpc Prune (PruneRequest) returns (PruneResponse);

}

message PruneRequest {
    // Inclusive offset up to which the ledger is to be pruned.
    string prune_up_to = 1;
}

message PruneResponse {
    // Empty for now, but may contain fields in the future
}
```

3.3.4.2 Domain Admin APIs

The domain exposes the following admin-api services:

Domain Status Service

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

syntax = "proto3";

package com.digitalasset.canton.health.admin.v0;
import "google/protobuf/empty.proto";
import "google/protobuf/duration.proto";
```

(continues on next page)

(continued from previous page)

```
import "google/protobuf/wrappers.proto";

service StatusService {
    rpc Status(google.protobuf.Empty) returns (NodeStatus);
}

message TopologyQueueStatus {
    // how many topology changes are currently queued at the manager
    uint32 manager = 1;
    // how many topology changes are currently queued at the dispatcher
    uint32 dispatcher = 2;
    // how many topology changes are currently waiting to become effective
↪(across all connected domains in the case of participants)
    uint32 clients = 3;
}

message NodeStatus {
    message Status {
        string id = 1;
        google.protobuf.Duration uptime = 2;
        map<string, int32> ports = 3;
        bytes extra = 4; // contains extra information depending on the node type
        bool active = 5; // Indicate if the node is active, usually true unless it
↪'s a replicated node that is passive
        TopologyQueueStatus topology_queues = 6; // indicates the state of the
↪topology queues (manager / dispatcher only where they exist)
    }

    message NotInitialized {
        bool active = 1; // Indicate if the node is active, usually true unless it
↪'s a replicated node that is passive
    }

    oneof response {
        NotInitialized not_initialized = 1; // node is running but has not been
↪initialized yet
        Status success = 2; // successful response from a running and initialized
↪node
    }
}



// domain node specific extra status info
message DomainStatusInfo {
    repeated string connected_participants = 1;
    // optional - only set if a sequencer is being run by the domain
    SequencerHealthStatus sequencer = 2;
}

// participant node specific extra status info
message ParticipantStatusInfo {
    message ConnectedDomain {
        string domain = 1;
        bool healthy = 2;
    }
```

(continues on next page)

(continued from previous page)

```
    repeated ConnectedDomain connected_domains = 1;
    // Indicate if the participant node is active
    // True if the participant node is replicated and is the active replica, or
↪true if not replicated
    bool active = 2;
}

message SequencerNodeStatus {
    repeated string connected_participants = 1;
    // required - status of the sequencer component it is running
    SequencerHealthStatus sequencer = 2;
}

// status of the sequencer component
message SequencerHealthStatus {
    // is the sequencer component active - can vary by implementation for what
↪this means
    bool active = 1;
    // optionally set details on how sequencer is healthy/unhealthy
    google.protobuf.StringValue details = 2;
}
```

3.3.4.3 Identity Admin APIs

Both, domain and participant nodes expose the following services:

Vault Management Service

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

syntax = "proto3";

package com.digitalasset.canton.crypto.admin.v0;

import "com/digitalasset/canton/crypto/v0/crypto.proto";

/**
 * Vault service providing programmatic access to locally stored keys and
↪certificates
 *
 * We have two key-stores: a private key store where we are storing our pairs of
 * public and private keys and a public key store where we are storing other
 * public keys that we learned.
 *
 * We learn public key stores in different ways: either by importing them or
 * by picking them up from internal sources (such as identity management updates).
 *
 * The only purpose of the public key store (where we import foreign keys) is
↪convenience for
 * identity management such that when we add identity management transactions, we
↪can refer to
```

(continues on next page)

(continued from previous page)

```
 * fingerprints in commands while building them rather than having to attach
↪public-key files.
 *
 * In addition, we also provide access to the locally stored certificates which
↪are used
 * either by the HTTP/1.1 sequencer client or for legal identity claims.
 */
service VaultService {

    /**
     * List public keys according to request filter for which we have a private
↪key in our key vault.
     *
     * The request includes a filter for fingerprints which can be used for
↪lookups.
     *
     * @param ListMyKeysRequest: request with optional fingerprint filter
     * @return: all serialized keys and their fingerprints that have the
↪fingerprint filter as a substring in their fingerprint
     */
    rpc ListMyKeys(ListKeysRequest) returns (ListKeysResponse);

    /**
     * Generates a new public / private key pair for signing.
     *
     * Stores the private key in the vault, and returns the public key
     */
    rpc GenerateSigningKey(GenerateSigningKeyRequest) returns
↪(GenerateSigningKeyResponse);

    /**
     * Generates a new public / private key pair for hybrid encryption.
     *
     * Stores the private key in the vault, and returns the public key
     */
    rpc GenerateEncryptionKey(GenerateEncryptionKeyRequest) returns
↪(GenerateEncryptionKeyResponse);

    /**
     * Import a public key into the registry in order to provide that Fingerprint
↪-> PublicKey lookups
     *
     * @param: ImportPublicKeyRequest serialized public key to be imported
     * @return: fingerprint and serialized public key of imported public key
     */
    rpc ImportPublicKey(ImportPublicKeyRequest) returns (ImportPublicKeyResponse);

    /**
     * Lists all public keys matching the supplied filter which are internally
↪cached
     *
     * Any public key returned here can be referenced in topology transaction
↪building
     * by fingerprint.
     */
    rpc ListPublicKeys(ListKeysRequest) returns (ListKeysResponse);
```

(continues on next page)

(continued from previous page)

```
    /**
     * Import a X509 certificate into the local vault.
     */
    rpc ImportCertificate(ImportCertificateRequest) returns
↪(ImportCertificateResponse);

    /**
     * Create a new, self-signed certificate with CN=unique_identifier
     */
    rpc GenerateCertificate(GenerateCertificateRequest) returns
↪(GenerateCertificateResponse);

    /**
     * List certificates stored in the local vault
     */
    rpc ListCertificates(ListCertificateRequest) returns
↪(ListCertificateResponse);

}

message GenerateCertificateRequest {
    // unique identifier to be used for CN
    string unique_identifier = 1;
    // the private key fingerprint to use for this certificate
    string certificate_key = 2;
    // optional additional X500 names
    string additional_subject = 3;
    // the additional subject names to be added to this certificate
    repeated string subject_alternative_names = 4;
}

message GenerateCertificateResponse {
    // the certificate in PEM format
    string x509_cert = 1;
}

message ListCertificateRequest {
    string filterUid = 1;
}

message ListCertificateResponse {
    message Result {
        string x509_cert = 1;
    }
    repeated Result results = 1;
}

message ImportCertificateRequest {
    // X509 certificate as PEM
    string x509_cert = 1;
}

message ImportCertificateResponse {
    string certificate_id = 1;
}
```

(continues on next page)

(continued from previous page)

```
message ImportPublicKeyRequest {
    // import a crypto.PublicKey protobuf serialized key
    bytes public_key = 1;
    // an optional name that should be stored along side the key
    string name = 2;
}

message ImportPublicKeyResponse {
    // fingerprint of imported key
    string fingerprint = 1;
}

message ListKeysRequest {
    // the substring that needs to match a given fingerprint
    string filter_fingerprint = 1;
    // the substring to filter the name
    string filter_name = 2;
    // filter on public key purpose
    repeated com.digitalasset.canton.crypto.v0.KeyPurpose filter_purpose = 3;
}

message ListKeysResponse {
    repeated com.digitalasset.canton.crypto.v0.PublicKeyWithName public_keys = 1;
}

message GenerateSigningKeyRequest {
    com.digitalasset.canton.crypto.v0.SigningKeyScheme key_scheme = 1;

    // optional descriptive name for the key
    string name = 2;
}

message GenerateSigningKeyResponse {
    com.digitalasset.canton.crypto.v0.SigningPublicKey public_key = 1;
}

message GenerateEncryptionKeyRequest {
    com.digitalasset.canton.crypto.v0.EncryptionKeyScheme key_scheme = 1;

    // optional descriptive name for the key
    string name = 2;
}

message GenerateEncryptionKeyResponse {
    com.digitalasset.canton.crypto.v0.EncryptionPublicKey public_key = 1;
}
```

Initialization Service

The one time initialization service, used to setup the identity of a node.

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

syntax = "proto3";

package com.digitalasset.canton.topology.admin.v0;

import "google/protobuf/empty.proto";
import "google/protobuf/timestamp.proto";

/**
 * The node initialization service
 */
service InitializationService {

    /**
     * Initialize the node with the unique identifier (can and must be done once)
     *
     * When a domain or participant instance starts for the first time, we need
↪to bind it
     * to a globally unique stable identifier before we can continue with the
     * initialization procedure.
     *
     * This method is only used once during initialization.
     */
    rpc InitId(InitIdRequest) returns (InitIdResponse);


    /**
     * Returns the id of the node (or empty if not initialized)
     */
    rpc GetId(google.protobuf.Empty) returns (GetIdResponse);


    /**
     * Returns the current time of the node (used for testing with static time)
     */
    rpc CurrentTime(google.protobuf.Empty) returns (google.protobuf.Timestamp);

}

message InitIdRequest {
    string identifier = 1;
    string fingerprint = 2;
    // optional - instance id, if supplied value is empty then one will be
↪generated
    string instance = 3;
}

message InitIdResponse {
    string unique_identifier = 1;
    string instance = 2;
```

(continues on next page)

(continued from previous page)

```
}

message GetIdResponse {
    bool initialized = 1;
    string unique_identifier = 2;
    string instance = 3;
}
```

Topology Aggregation Service

Aggregated view of the sequenced domain topology state.

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

syntax = "proto3";

package com.digitalasset.canton.topology.admin.v0;

import "google/protobuf/timestamp.proto";
import "com/digitalasset/canton/crypto/v0/crypto.proto";
import "com/digitalasset/canton/protocol/v0/topology.proto";

/**
 * Topology information aggregation service
 *
 * This service allows deep inspection into the aggregated topology state.
 * The services runs both on the domain and on the participant and uses the same
 * data. The service provides GRPC access to the information aggregated by the
↪identity providing
 * service client.
 */
service TopologyAggregationService {

    /**
      * Obtain information about a certain set of active parties matching a given
↪filter criterion.
      *
      * The request allows to filter per (domain, party, asOf) where the domain
↪and party argument are
      * used in order to filter the result list using the `startsWith` method on
↪the respective resulting string.
      *
      * As such, if you just need to do a lookup, then define a precise filter.
↪Given the uniqueness of the
      * identifiers (and the fact that the identifiers contain key fingerprints),
↪we should only ever get a single
      * result back if we are doing a precise lookup.
      *
      * The response is a sequence of tuples (party, domain, participant,
↪privilege, trust-level).
      * The response is restricted to active parties and their active
↪participants.
```

(continues on next page)

(continued from previous page)

```
      */
    rpc ListParties (ListPartiesRequest) returns (ListPartiesResponse);

    /**
      * Obtain key owner information matching a given filter criterion.
      *
      * Key owners in the system are different types of entities: Participant,
↪Mediator, Domain Topology Manager and
      * Sequencer. The present method allows to define a filter to search for a
↪key owner
      * using filters on (asOf, domain, ownerType, owner)
      *
      * The response is a sequence of (domain, ownerType, owner, keys) where keys
↪is a sequence of
      * (fingerprint, bytes, key purpose). As such, we can use this method to
↪export currently used signing or encryption
      * public keys.
      *
      * This method is quite general, as depending on the arguments, very
↪different results can be obtained.
      *
      * Using OwnerType = 'Participant' allows to query for all participants.
      * Using OwnerType = 'Sequencer' allows to query for all sequencers defined.
      */
    rpc ListKeyOwners (ListKeyOwnersRequest) returns (ListKeyOwnersResponse);
}

message ListPartiesRequest {
    google.protobuf.Timestamp as_of = 1;
    int32 limit = 2;
    string filter_domain = 3;
    string filter_party = 4;
    string filter_participant = 5;
}

message ListPartiesResponse {
    message Result {
        string party = 1;
        message ParticipantDomains {
            message DomainPermissions {
                string domain = 1;
                com.digitalasset.canton.protocol.v0.ParticipantPermission
↪permission = 2;
            }
            string participant = 1;
            /**
              * permissions of this participant for this party on a per domain
↪basis
              *
              * for records that only exist in the authorized store, this list
↪will be empty.
              */
            repeated DomainPermissions domains = 2;
        }
        repeated ParticipantDomains participants = 2;
    }
```

(continues on next page)

(continued from previous page)

```
    repeated Result results = 2;
}

message ListKeyOwnersRequest {
    google.protobuf.Timestamp as_of = 1;
    int32 limit = 2;
    string filter_domain = 3;
    string filter_key_owner_type = 4;
    string filter_key_owner_uid = 5;
}

message ListKeyOwnersResponse {
    message Result {
        string domain = 1;
        string key_owner = 2;
        repeated com.digitalasset.canton.crypto.v0.SigningPublicKey signing_keys
= 3;
        repeated com.digitalasset.canton.crypto.v0.EncryptionPublicKey encryption_
keys = 4;
    }
    repeated Result results = 1;
}
```

Topology Manager Read Service

Raw access to the underlying topology transactions.

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

syntax = "proto3";

package com.digitalasset.canton.topology.admin.v0;

import "com/digitalasset/canton/crypto/v0/crypto.proto";
import "com/digitalasset/canton/protocol/v0/topology.proto";
import "com/digitalasset/canton/protocol/v0/sequencing.proto";
import "google/protobuf/timestamp.proto";
import "google/protobuf/empty.proto";

// domain + idm + participant
service TopologyManagerReadService {

    rpc ListAvailableStores(ListAvailableStoresRequest) returns
(ListAvailableStoresResult);
    rpc ListPartyToParticipant(ListPartyToParticipantRequest) returns
(ListPartyToParticipantResult);
    rpc ListOwnerToKeyMapping(ListOwnerToKeyMappingRequest) returns
(ListOwnerToKeyMappingResult);
    rpc ListNamespaceDelegation(ListNamespaceDelegationRequest) returns
(ListNamespaceDelegationResult);
    rpc ListIdentifierDelegation(ListIdentifierDelegationRequest) returns
(ListIdentifierDelegationResult);
```

(continues on next page)

(continued from previous page)

```
    rpc ListSignedLegalIdentityClaim(ListSignedLegalIdentityClaimRequest) returns
↪(ListSignedLegalIdentityClaimResult);
    rpc ListParticipantDomainState(ListParticipantDomainStateRequest) returns
↪(ListParticipantDomainStateResult);
    rpc ListMediatorDomainState(ListMediatorDomainStateRequest) returns
↪(ListMediatorDomainStateResult);
    rpc ListVettedPackages(ListVettedPackagesRequest) returns
↪(ListVettedPackagesResult);
    rpc ListDomainParametersChanges(ListDomainParametersChangesRequest) returns
↪(ListDomainParametersChangesResult);
    rpc ListAll(ListAllRequest) returns (ListAllResponse);


}

message ListNamespaceDelegationRequest {
    BaseQuery base_query = 1;
    string filter_namespace = 2;
}

message ListNamespaceDelegationResult {
    message Result {
        BaseResult context = 1;
        com.digitalasset.canton.protocol.v0.NamespaceDelegation item = 2;
        string target_key_fingerprint = 3;
    }
    repeated Result results = 1;
}

message ListIdentifierDelegationRequest {
    BaseQuery base_query = 1;
    string filter_uid = 2;
}

message ListIdentifierDelegationResult {
    message Result {
        BaseResult context = 1;
        com.digitalasset.canton.protocol.v0.IdentifierDelegation item = 2;
        string target_key_fingerprint = 3;
    }
    repeated Result results = 1;
}

message BaseQuery {
    string filter_store = 1;
    bool use_state_store = 2;
    com.digitalasset.canton.protocol.v0.TopologyChangeOp operation = 3;
    /** if true, then we'll filter the results according to above defined
↪operation */
    bool filter_operation = 4;
    message TimeRange {
        google.protobuf.Timestamp from = 2;
        google.protobuf.Timestamp until = 3;
    }
    oneof time_query {
        google.protobuf.Timestamp snapshot = 5;
```

(continues on next page)

(continued from previous page)

```
        google.protobuf.Empty head_state = 6;
        TimeRange range = 7;
    }
    string filter_signed_key = 8;
}

message BaseResult {
    string store = 1;
    google.protobuf.Timestamp valid_from = 2;
    google.protobuf.Timestamp valid_until = 3;
    com.digitalasset.canton.protocol.v0.TopologyChangeOp operation = 4;
    bytes serialized = 5;
    string signed_by_fingerprint = 6;
}

message ListPartyToParticipantResult {
    message Result {
        BaseResult context = 1;
        com.digitalasset.canton.protocol.v0.PartyToParticipant item = 2;
    }
    repeated Result results = 2;
}

message ListPartyToParticipantRequest {
    BaseQuery base_query = 1;
    string filter_party = 2;
    string filter_participant = 3;
    message FilterRequestSide {
        com.digitalasset.canton.protocol.v0.RequestSide value = 1;
    }
    FilterRequestSide filter_request_side = 4;
    message FilterPermission {
        com.digitalasset.canton.protocol.v0.ParticipantPermission value = 1;
    }
    FilterPermission filter_permission = 5;
}

message ListOwnerToKeyMappingRequest {
    BaseQuery base_query = 1;
    string filter_key_owner_type = 2;
    string filter_key_owner_uid = 3;
    message FilterKeyPurpose {
        com.digitalasset.canton.crypto.v0.KeyPurpose value = 1;
    }
    FilterKeyPurpose filter_key_purpose = 4;
}

message ListOwnerToKeyMappingResult {
    message Result {
        BaseResult context = 1;
        com.digitalasset.canton.protocol.v0.OwnerToKeyMapping item = 2;
        string key_fingerprint = 3;
    }
    repeated Result results = 1;
}
```

(continues on next page)

(continued from previous page)

```
message ListSignedLegalIdentityClaimRequest {
    BaseQuery base_query = 1;
    string filter_uid = 2;
}

message ListSignedLegalIdentityClaimResult {
    message Result {
        BaseResult context = 1;
        com.digitalasset.canton.protocol.v0.SignedLegalIdentityClaim item = 2;
    }
    repeated Result results = 1;
}

message ListVettedPackagesRequest {
    BaseQuery base_query = 1;
    string filter_participant = 2;
}

message ListVettedPackagesResult {
    message Result {
        BaseResult context = 1;
        com.digitalasset.canton.protocol.v0.VettedPackages item = 2;
    }
    repeated Result results = 1;
}

message ListDomainParametersChangesRequest {
    BaseQuery base_query = 1;
}

message ListDomainParametersChangesResult {
    message Result {
        BaseResult context = 1;
        com.digitalasset.canton.protocol.v0.DynamicDomainParameters item = 2;
    }
    repeated Result results = 1;
}

message ListAvailableStoresRequest {

}
message ListAvailableStoresResult {
    repeated string store_ids = 1;
}

message ListParticipantDomainStateRequest {
    BaseQuery base_query = 1;
    string filter_domain = 2;
    string filter_participant = 3;
}

message ListParticipantDomainStateResult {
    message Result {
        BaseResult context = 1;
        com.digitalasset.canton.protocol.v0.ParticipantState item = 2;
    }
```

(continues on next page)

(continued from previous page)

```
    repeated Result results = 1;
}

message ListMediatorDomainStateRequest {
    BaseQuery base_query = 1;
    string filter_domain = 2;
    string filter_mediator = 3;
}

message ListMediatorDomainStateResult {
    message Result {
        BaseResult context = 1;
        com.digitalasset.canton.protocol.v0.MediatorDomainState item = 2;
    }
    repeated Result results = 1;
}


message ListAllRequest {
    BaseQuery base_query = 1;
}

message ListAllResponse {
    com.digitalasset.canton.protocol.v0.TopologyTransactions result = 1;
}
```

Topology Manager Write Service

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

syntax = "proto3";

package com.digitalasset.canton.topology.admin.v0;

import "com/digitalasset/canton/crypto/v0/crypto.proto";
import "com/digitalasset/canton/protocol/v0/topology.proto";
import "com/digitalasset/canton/protocol/v0/sequencing.proto";

/**
 * Write operations on the local topology manager.
 *
 * Both, participant and domain run a local topology manager exposing the same
↪write interface.
 */
service TopologyManagerWriteService {

    /**
     * Authorizes a party to participant mapping change (add/remove) on the node
↪local topology manager.
     */
    rpc AuthorizePartyToParticipant(PartyToParticipantAuthorization) returns
↪(AuthorizationSuccess);
```

(continues on next page)

(continued from previous page)

```
    /**
     * Authorizes an owner to key mapping change (add/remove) on the local
↪topology manager
     */
    rpc AuthorizeOwnerToKeyMapping(OwnerToKeyMappingAuthorization) returns
↪(AuthorizationSuccess);

    /**
     * Authorizes a namespace delegation (root or intermediate CA) (add/remove)
↪on the local topology manager
     */
    rpc AuthorizeNamespaceDelegation(NamespaceDelegationAuthorization) returns
↪(AuthorizationSuccess);

    /**
     * Authorizes a new identifier delegation (identifier certificate) (add/
↪remove) on the local topology manager
     */
    rpc AuthorizeIdentifierDelegation(IdentifierDelegationAuthorization) returns
↪(AuthorizationSuccess);

    /**
     * Authorizes a new package vetting transaction
     */
    rpc AuthorizeVettedPackages(VettedPackagesAuthorization) returns
↪(AuthorizationSuccess);

    /** Authorizes a change of parameters for the domain */
    rpc AuthorizeDomainParametersChange(DomainParametersChangeAuthorization)
↪returns (AuthorizationSuccess);

    /**
     * Authorizes a new signed legal identity
     */
    rpc AuthorizeSignedLegalIdentityClaim(SignedLegalIdentityClaimAuthorization)
↪returns (AuthorizationSuccess);

    /**
     * Authorizes a participant domain state
     */
    rpc AuthorizeParticipantDomainState(ParticipantDomainStateAuthorization)
↪returns (AuthorizationSuccess);

    /**
     * Authorizes a mediator domain state
     */
    rpc AuthorizeMediatorDomainState(MediatorDomainStateAuthorization) returns
↪(AuthorizationSuccess);

    /**
     * Adds a signed topology transaction to the Authorized store
     */
    rpc AddSignedTopologyTransaction(SignedTopologyTransactionAddition) returns
↪(AdditionSuccess);
```

(continues on next page)

(continued from previous page)

```
    /**
     * Generates a legal identity claim
     */
    rpc GenerateSignedLegalIdentityClaim(SignedLegalIdentityClaimGeneration)
↪returns (com.digitalasset.canton.protocol.v0.SignedLegalIdentityClaim);

}

message AuthorizationSuccess {
    bytes serialized = 1;
}

message AdditionSuccess {

}

message SignedTopologyTransactionAddition {
    bytes serialized = 1;
}

message AuthorizationData {

    /** Add / Remove / Replace */
    com.digitalasset.canton.protocol.v0.TopologyChangeOp change = 1;

    /**
     * Fingerprint of the key signing the authorization
     *
     * The signing key is used to identify a particular `NamespaceDelegation` or
↪`IdentifierDelegation` certificate,
     * which is used to justify the given authorization.
     */
    string signed_by = 2;

    /** if true, the authorization will also replace the existing (makes only
↪sense for adds) */
    bool replace_existing = 3;

    /** Force change even if dangerous */
    bool force_change = 4;

}

message NamespaceDelegationAuthorization {
    AuthorizationData authorization = 1;

    // The namespace for which the authorization is issued.
    string namespace = 2;

    /**
     * The fingerprint of the signing key which will be authorized to issue
↪topology transactions for this namespace.
     *
     * The key needs to be present in the local key registry either by being
↪locally
     * generated or by having been previously imported.
```

(continues on next page)

(continued from previous page)

```
     */
    string fingerprint_of_authorized_key = 3;

    /**
     * Flag indicating whether authorization is a root key delegation
     */
    bool is_root_delegation = 4;
}

message IdentifierDelegationAuthorization {
    AuthorizationData authorization = 1;
    string identifier = 2;

    /**
     * The fingerprint of the signing key which will be authorized to issue
↪topology transaction for this particular identifier.
     *
     * As with `NamespaceDelegation`s, the key needs to be present locally.
     */
    string fingerprint_of_authorized_key = 3;
}

message PartyToParticipantAuthorization {
    AuthorizationData authorization = 1;
    /**
     * The request side of this transaction
     *
     * A party to participant mapping can map a party from one namespace on a
↪participant from another namespace.
     * Such a mapping needs to be authorized by both namespace keys. If the
↪namespace is the same, we use
     * RequestSide.Both and collapse into a single transaction. Otherwise, `From`
↪needs to be signed by a namespace key
     * of the party and `To` needs to be signed by a namespace key of the
↪participant.
     */
    com.digitalasset.canton.protocol.v0.RequestSide side = 2;

    // The unique identifier of the party
    string party = 3;
    // The unique identifier of the participant
    string participant = 4;
    // The permission of the participant that will allow him to act on behalf of
↪the party.
    com.digitalasset.canton.protocol.v0.ParticipantPermission permission = 5;
}

message OwnerToKeyMappingAuthorization {

    AuthorizationData authorization = 1;

    /**
     * The key owner
     *
     * An entity in Canton is described by his role and his unique identifier. As
↪such, the same unique identifier
```

(continues on next page)

(continued from previous page)

```
     * can be used for a mediator, sequencer, domain topology manager or even
↪participant. Therefore, we expect
     * here the protoPrimitive of a key owner which is in effect its type as a
↪three letter code separated
     * from the unique identifier.
     */
    string key_owner = 2;


    /**
     * The fingerprint of the key that will be authorized
     *
     * The key needs to be present in the local key registry (can be imported via
↪KeyService)
     */
    string fingerprint_of_key = 3;

    /**
     * Purpose of the key
     */
    com.digitalasset.canton.crypto.v0.KeyPurpose key_purpose = 4;

}

message SignedLegalIdentityClaimAuthorization {
    AuthorizationData authorization = 1;
    com.digitalasset.canton.protocol.v0.SignedLegalIdentityClaim claim = 2;
}

message SignedLegalIdentityClaimGeneration {
    message X509CertificateClaim {
        string unique_identifier = 1;
        string certificate_id = 2;
    }
    oneof request {
        // Serialized LegalIdentityClaim
        bytes legal_identity_claim = 1;
        X509CertificateClaim certificate = 2;
    }
}


message ParticipantDomainStateAuthorization {
    AuthorizationData authorization = 1;
    /** which side (domain or participant) is attempting to issue the
↪authorization */
    com.digitalasset.canton.protocol.v0.RequestSide side = 2;
    /** domain this authorization refers to */
    string domain = 3;
    /** participant that should be authorized */
    string participant = 4;
    /** permission that should be used (lower of From / To) */
    com.digitalasset.canton.protocol.v0.ParticipantPermission permission = 5;
    /** trust level that should be used (ignored for side from, defaults to
↪Ordinary) */
    com.digitalasset.canton.protocol.v0.TrustLevel trust_level = 6;
```

(continues on next page)

(continued from previous page)

```
}

message MediatorDomainStateAuthorization {
    AuthorizationData authorization = 1;
    /** which side (domain or mediator) is attempting to issue the authorization
↪*/
    com.digitalasset.canton.protocol.v0.RequestSide side = 2;
    /** domain this authorization refers to */
    string domain = 3;
    /** mediator that should be authorized */
    string mediator = 4;
}

message VettedPackagesAuthorization {
    AuthorizationData authorization = 1;
    string participant = 2;
    repeated string package_ids = 3;
}

message DomainParametersChangeAuthorization {
    AuthorizationData authorization = 1;
    /** domain this authorization refers to */
    string domain = 2;
    /** new parameters for the domain */
    com.digitalasset.canton.protocol.v0.DynamicDomainParameters parameters = 3;
}
```

3.3.4.4 Mediator Admin APIs

Standalone Mediator nodes (enterprise version only) expose the following services:

Mediator Initialization Service

Service to initialize an external Mediator to participate in confirming transaction results. Only expected to be called by the Domain node to allow the Mediator to connect to the domain Sequencer.

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

syntax = "proto3";

package com.digitalasset.canton.domain.admin.v0;

import "com/digitalasset/canton/crypto/v0/crypto.proto";
import "com/digitalasset/canton/protocol/v0/sequencing.proto";
import "com/digitalasset/canton/protocol/v0/topology.proto";


service MediatorInitializationService {

  // Initialize a Mediator service
```

(continues on next page)

(continued from previous page)

```
  // If the Mediator is uninitialized it should initialize itself with the
↪provided configuration
  // If the Mediator is already initialized then verify the request is for the
↪domain we're running against,
  // if correct then just return the current key otherwise fail.
  rpc Initialize (InitializeMediatorRequest) returns (InitializeMediatorResponse);

}

message InitializeMediatorRequest {
  // the domain identifier
  string domain_id = 1;
  // the mediator identifier
  string mediator_id = 2;
  // topology state required for startup
  com.digitalasset.canton.protocol.v0.TopologyTransactions current_identity_state
↪= 3;
  // parameters for the domain (includes the protocol version which needs to
↪match the protocol version the domain
  // manager is running)
  com.digitalasset.canton.protocol.v0.StaticDomainParameters domain_parameters =
↪4;
  // how should the member connect to the domain sequencer
  com.digitalasset.canton.protocol.v0.SequencerConnection sequencer_connection =
↪5;
}

message InitializeMediatorResponse {
  oneof value {
    Success success = 1;
    Failure failure = 2;
  }

  message Success {
    // Current signing key
    com.digitalasset.canton.crypto.v0.SigningPublicKey mediator_key = 1;
  }

  message Failure {
    // Reason that can be logged
    string reason = 1;
  }
}
```

Enterprise Mediator Administration Service

Important: This feature is only available in Canton Enterprise

Exposes details about the mediator operation such as its leadership status when many mediator instances are running in a single domain to provide high availability.

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

syntax = "proto3";

package com.digitalasset.canton.domain.admin.v0;

import "google/protobuf/timestamp.proto";
import "google/protobuf/empty.proto";

// administration service for mediator instances
service EnterpriseMediatorAdministrationService {

  // Remove unnecessary data from the Mediator
  rpc Prune (MediatorPruningRequest) returns (google.protobuf.Empty);

}

message MediatorPruningRequest {
  // timestamp to prune for
  google.protobuf.Timestamp timestamp = 1;
}
```

3.3.4.5 Sequencer Admin APIs

Standalone Sequencer nodes (enterprise version only) expose the following services:

Sequencer Administration Service

Important: This feature is only available in Canton Enterprise

Exposes status information of the Sequencer.

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

syntax = "proto3";

package com.digitalasset.canton.domain.admin.v0;

import "google/protobuf/empty.proto";
import "google/protobuf/timestamp.proto";

// administration service for sequencer instances
service SequencerAdministrationService {

  // fetch the current status of the sequencer
  rpc PruningStatus (google.protobuf.Empty) returns (SequencerPruningStatus);
```

(continues on next page)

(continued from previous page)

```
}

message SequencerMemberStatus {
  string member = 1;
  google.protobuf.Timestamp registered_at = 2;
  google.protobuf.Timestamp last_acknowledged = 3;
  bool enabled = 4;
}

message SequencerPruningStatus {
  // current time according to the sequencer
  google.protobuf.Timestamp now = 1;
  // the earliest event we are currently storing
  google.protobuf.Timestamp earliest_event_timestamp = 2;
  // details of each member registered on the sequencer
  repeated SequencerMemberStatus members = 3;
}
```

Enterprise Sequencer Administration Service

Exposes enterprise features of the Sequencer, such as pruning and the ability to disable clients.

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

syntax = "proto3";

package com.digitalasset.canton.domain.admin.v0;

import "google/protobuf/empty.proto";
import "google/protobuf/timestamp.proto";
import "com/digitalasset/canton/domain/admin/v0/sequencer_initialization_service.
↪proto";

// administration service for enterprise feature supporting sequencer instances
service EnterpriseSequencerAdministrationService {

  // Remove data from the Sequencer
  rpc Prune (Pruning.Request) returns (Pruning.Response);

  // fetch a snapshot of the sequencer state based on the given timestamp
  rpc Snapshot(Snapshot.Request) returns (Snapshot.Response);

  // Disable members at the sequencer. Will prevent existing and new instances
↪from connecting, and permit removing their data.
  rpc DisableMember(DisableMemberRequest) returns (google.protobuf.Empty);
}

message Pruning {
  message Request {
    google.protobuf.Timestamp timestamp = 1;
  }
```

(continues on next page)

(continued from previous page)

```
  message Response {
    // description of what was removed
    string details = 1;
  }
}

message Snapshot {
  message Request {
    google.protobuf.Timestamp timestamp = 1;
  }
  message Response {
    oneof value {
      Success success = 1;
      Failure failure = 2;
    }
  }
  message Success {
    com.digitalasset.canton.domain.admin.v0.SequencerSnapshot state = 1;
  }
  message Failure {
    string reason = 1;
  }
}

message DisableMemberRequest {
  string member = 1;
}
```

3.3.5 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.

3.3.5.1 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.

3.3.5.2 Run Modes

Canton can run in three different modes, depending on the desired environment and task.

Interactive Console

The default and recommended method to run Canton is in the interactive mode. This is the mode Canton will start in by default. 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 `CommandExcecutionException` 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.

This is the recommended way to run Canton (for now).

For server use on Linux / OSX, we recommend to run the application 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.

Remote Console Mode

You can also run the console process separate from the participant or domain nodes. Some advanced console commands (e.g. for testing) that require in-process access to the node will not be available, but all commands that run over the administrative GRPC APIs will work.

Running the console on the remote node requires a separate, albeit limited configuration with the information on how to connect to the admin and ledger-api.

For a participant, you need something like

```
canton {
  remote-participants {
    remoteParticipant1 {
      admin-api {
        port = 10012
        address = 127.0.0.1 // is the default value if omitted
      }
      ledger-api {
        port = 10011
        address = 127.0.0.1 // is the default value if omitted
```

(continues on next page)

(continued from previous page)

```
      }
    }
  }
}
```

whereas for a domain, a configuration would look like

```
canton {
  remote-domains {
    remoteDomain1 {
      public-api {
        address = 127.0.0.1
        port = 10018
      }
      admin-api {
        port = 10019
        address = 127.0.0.1 // default value if omitted
      }
    }
  }
}
```

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.

This mode is sometimes useful for testing, but we are not convinced yet that we'll keep it in a stable version.

Daemon

If the console is undesired, Canton can be started in daemon mode

```
bin/canton daemon --config ...
```

All configured entities will be automatically started and will resume operation. Any failures encountered during start up will immediately shutdown the Canton process with a non-zero exit code. This mode is interesting if a third party administration tool is used with Canton.

3.3.5.3 Flush Log Files Immediately

By default, Canton will immediately flush log output to the log file so that nothing is lost in case of a crash. To get the best possible throughput, you can switch this off by running Canton with `--log-immediate-flush false`.

3.3.5.4 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. For example, you can configure the heap size as follows:

```
JAVA_OPTS="-Xmx2G" ./bin/canton --config ...
```

3.3.6 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

- *Remote Administration*
- *Node References*
- *Help*
- *Lifecycle Operations*
- *Timeouts*
- *Other Top-level Commands*
- *Participant Commands*
  - *Database*
  - *Health*
  - *Domain Connectivity*
  - *Packages*
  - *DAR Management*
  - *DAR Sharing*
  - *Party Management*
  - *Key Administration*
  - *Topology Administration*
  - *Ledger API Access*
    - *Transaction Service*
    - *Command Service*
    - *Command Completion Service*

        - Active Contract Service
        - Package Service
        - Party Management Service
        - Ledger Configuration Service
        - Ledger Api User Management Service
        - Ledger Api Metering Service
    - Composability
    - Ledger Pruning
    - Bilateral Commitments
    - Participant Repair
    - Resource Management
    - Replication
  Domain Administration Commands
    - Health
    - Database
    - Participants
    - Sequencer
    - Mediator
    - Key Administration
    - Parties
    - Service
    - Topology Administration
  Domain Manager Administration Commands
    - Setup
    - Health
    - Database
    - Key Administration
    - Parties
    - Service
    - Topology Administration
  Sequencer Administration Commands
    - Sequencer
    - Health
    - Database
  Mediator Administration Commands
    - Mediator
    - Health
    - Database
  Code-Generation in Console

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.

3.3.6.1 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
        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.

3.3.6.2 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`. Additionally, the console provides the following generic references.

participant1

Summary: Manage participant 'participant1'; type 'participant1 help' or 'participant1 help(<methodName>)' for more help

participant2
: **Summary**: Manage participant 'participant2'; type 'participant2 help' or 'participant2 help(<methodName>)' for more help

mydomain
: **Summary**: Manage local domain 'mydomain'; type 'mydomain help' or 'mydomain help(<methodName>)' for more help

domainManager1
: **Summary**: Manage local manager 'domainManager1'; type 'domainManager1 help' or 'domainManager1 help(<methodName>)' for more help

sequencer1
: **Summary**: Manage local sequencer 'sequencer1'; type 'sequencer1 help' or 'sequencer1 help(<methodName>)' for more help

mediator1
: **Summary**: Manage local mediator 'mediator1'; type 'mediator1 help' or 'mediator1 help(<methodName>)' for more help

participants
: **Summary**: All participant nodes

domains
: **Summary**: All domain nodes

nodes
: **Summary**: All nodes

sequencers
: **Summary**: All sequencer nodes

mediators
: **Summary**: All mediator nodes

domainManagers
: **Summary**: All domain manager nodes

3.3.6.3 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")`).

3.3.6.4 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 run manually using the command. In some rare cases, it may also be necessary to run `db.repair_migration` before running `db.migration` - please refer to the description of `db.repair_migration` for more details. 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.

3.3.6.5 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 `TimeoutDuration`. 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 `TimeoutDuration`. 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.

3.3.6.6 Other Top-level Commands

The following commands are available for convenience:

help

Summary: Help with console commands; type help(<command>) for detailed help for <command>

exit

Summary: Leave the console

health.dump

Summary: Generate and write a dump of Canton's state for a bug report
Return type:
- String

health.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.TimeoutDuration

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.

console.command_timeout

Summary: Yields the timeout for running console commands
Return type:
- com.digitalasset.canton.config.TimeoutDuration

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

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.get_level

Summary: Determine current logging level

Arguments:

- `loggerName`: String

Return type:

- Option[ch.qos.logback.classic.Level]

logging.set_level

Summary: Dynamically change log level (TRACE, DEBUG, INFO, WARN, ERROR, OFF, null)

Arguments:

- `loggerName`: String
- `level`: String

logging.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

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.write_to_file

Summary: Writes a ByteString to a file.

Arguments:

- `data`: com.google.protobuf.ByteString
- `fileName`: String

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.write_to_file

Summary: Writes a Protobuf message to a file.

Arguments:

- `data`: scalapb.GeneratedMessage
- `fileName`: String

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.write_to_file

Summary: Writes several Protobuf messages to a file.

Arguments:

- `data`: Seq[scalapb.GeneratedMessage]
- `fileName`: String

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.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.auto_close (Testing)

Summary: Register *AutoCloseable* object to be shutdown if Canton is shut down

Arguments:

- `closeable`: AutoCloseable

utils.generate_daml_script_participants_conf

Summary: Create a participants config for Daml script

Arguments:

- `file`: Option[String]
- `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:

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.retry_until_true

Summary: Wait for a condition to become true

Arguments:

- `timeout`: com.digitalasset.canton.config.TimeoutDuration
- `maxWaitPeriod`: com.digitalasset.canton.config.TimeoutDuration
- `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.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.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.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

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.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.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:
- `methodName`: String

3.3.6.7 Participant Commands

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.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.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.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`: Option[Int]

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.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`: Option[Int]

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.

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`: Option[Int]

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.acs_search (Testing)

Summary: Lookup of active contracts

Arguments:

- `domainAlias`: com.digitalasset.canton.DomainAlias
- `filterId`: String
- `filterPackage`: String
- `filterTemplate`: String
- `limit`: Int

Return type:

- List[com.digitalasset.canton.protocol.SerializableContract]

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`: Int

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.await_domain_time (Testing)

Summary: Await for the given time to be reached on the given domain

Arguments:

- `domainId`: com.digitalasset.canton.DomainId
- `time`: com.digitalasset.canton.data.CantonTimestamp
- `timeout`: com.digitalasset.canton.config.TimeoutDuration

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.TimeoutDuration

testing.fetch_domain_times (Testing)

Summary: Fetch the current time from all connected domains

Arguments:

- `timeout`: com.digitalasset.canton.config.TimeoutDuration

testing.fetch_domain_time (Testing)

Summary: Fetch the current time from the given domain

Arguments:

- `domainId`: com.digitalasset.canton.DomainId
- `timeout`: com.digitalasset.canton.config.TimeoutDuration

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.TimeoutDuration

Return type:
- com.digitalasset.canton.data.CantonTimestamp

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.TimeoutDuration
- `levels`: Long
- `gracePeriodMillis`: Long
- `workflowId`: String
- `id`: String

Return type:
- Option[scala.concurrent.duration.Duration]

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.TimeoutDuration
- `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.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

config

Summary: Return participant config
Return type:
- com.digitalasset.canton.participant.config.LocalParticipantConfig

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

stop

Summary: Stop the instance

start

Summary: Start the instance

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

help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

Database

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.

db.migrate

Summary: Migrates the instance's database if using a database storage

db.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:
- `methodName`: String

Health

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.TimeoutDuration
- `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.TimeoutDuration
- `workflowId`: String
- `id`: String

Return type:
- scala.concurrent.duration.Duration

health.wait_for_initialized

Summary: Wait for the node to be initialized

health.wait_for_running

Summary: Wait for the node to be running

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.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:
- `methodName`: String

Domain Connectivity

domains.accept_agreement

Summary: Accept the service agreement of the given domain alias
Arguments:
- `domainAlias`: com.digitalasset.canton.DomainAlias
- `agreementId`: String

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.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.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.

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.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_registered

Summary: List the configured domains of this participant
Return type:
- Seq[(com.digitalasset.canton.participant.domain.DomainConnectionConfig, Boolean)]

domains.list_connected

Summary: List the connected domains of this participant
Return type:
- Seq[com.digitalasset.canton.admin.api.client.data.ListConnectedDomainsResult]

domains.disconnect_local

Summary: Disconnect this participant from the given local domain
Arguments:
- `domain`: com.digitalasset.canton.console.DomainReference

domains.disconnect

Summary: Disconnect this participant from the given domain
Arguments:
- `domainAlias`: com.digitalasset.canton.DomainAlias

domains.reconnect_all

Summary: Reconnect this participant to all domains which are not marked as manual start
Arguments:
- `ignoreFailures`: Boolean

Description: If ignoreFailures is set to true (default), the command will ignore domains we currenty can't connect and proceed with all other domains.

domains.reconnect_local

Summary: Reconnect this participant to the given local domain
Arguments:
- `ref`: com.digitalasset.canton.console.DomainReference
- `retry`: Boolean

Return type:
- Boolean

Description: Idempotent attempts to re-establish a connection to the given local domain. Same behaviour as generic reconnect.

domains.reconnect

Summary: Reconnect this participant to the given domain
Arguments:
- `domainAlias`: com.digitalasset.canton.DomainAlias
- `retry`: Boolean

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.

domains.connect_ha

Summary: 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: 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_ha(mydomain , sequencer1, sequencer2) or: connect_ha(mydomain , 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)

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.DomainId]
- `certificatesPath`: String
- `priority`: Int
- `timeTrackerConfig`: com.digitalasset.canton.time.DomainTimeTrackerConfig

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.

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_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

domains.is_connected (Testing)

Summary: Test whether a participant is connected to a domain reference

Arguments:

- `reference`: com.digitalasset.canton.console.commands.DomainAdministration

Return type:

- Boolean

domains.active (Testing)

Summary: Test whether a participant is connected to and permissioned on a domain reference

Arguments:

- `reference`: com.digitalasset.canton.console.commands.DomainAdministration

Return type:

- Boolean

domains.active

Summary: Test whether a participant is connected to and permissioned on a domain where we have a healthy subscription.

Arguments:

- `domain`: com.digitalasset.canton.DomainAlias

Return type:

- Boolean

domains.id_of

Summary: Returns the id of the given domain alias

Arguments:

- `domainAlias`: com.digitalasset.canton.DomainAlias

Return type:

- com.digitalasset.canton.DomainId

domains.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

Packages

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.TimeoutDuration

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.

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.find

Summary: Find packages that contain a module with the given name

Arguments:

- `moduleName`: String

Return type:

- Seq[com.digitalasset.canton.participant.admin.v0.PackageDescription]

packages.list_contents

Summary: List package contents

Arguments:

- `packageId`: String

Return type:

- Seq[com.digitalasset.canton.participant.admin.v0.ModuleDescription]

packages.list

Summary: List packages stored on the participant

Arguments:

- `limit`: Option[Int]

Return type:

- Seq[com.digitalasset.canton.participant.admin.v0.PackageDescription]

Description: If a limit is given, only up to *limit* packages are returned.

packages.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

DAR Management

dars.download

Summary: Downloads the DAR file with the given hash to the given directory
Arguments:
- `darHash`: String
- `directory`: String

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()

dars.list

Summary: List installed DAR files
Arguments:
- `limit`: Option[Int]

Return type:
- Seq[com.digitalasset.canton.participant.admin.v0.DarDescription]

dars.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

DAR Sharing

dars.sharing.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.requests.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

dars.sharing.offers.reject (Preview)

Summary: Reject the offer to share a DAR

Arguments:

- `shareId`: String
- `reason`: String

dars.sharing.offers.accept (Preview)

Summary: Accept the offer to share a DAR

Arguments:

- `shareId`: 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.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

dars.sharing.whitelist.remove (Preview)

Summary: Remove party from my DAR sharing whitelist

Arguments:

- `partyId`: com.digitalasset.canton.topology.PartyId

dars.sharing.whitelist.add (Preview)

Summary: Add party to my DAR sharing whitelist
Arguments:
- `partyId`: com.digitalasset.canton.topology.PartyId

dars.sharing.whitelist.list (Preview)
Summary: List parties that are currently whitelisted to share DARs with me

dars.sharing.whitelist.help
Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

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.TimeoutDuration

Description: Will throw an exception if the given topology has not been observed within the given timeout.

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.disable
Summary: Disable party on participant
Arguments:
- `name`: com.digitalasset.canton.topology.Identifier

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. Option-

ally, 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.hosted

Summary: List parties managed by this participant

Arguments:

- `filterParty`: String
- `filterDomain`: String
- `asOf`: Option[java.time.Instant]

Return type:

- Seq[com.digitalasset.canton.admin.api.client.data.ListPartiesResult]

Description: The filterDomain parameter is used to filter the results by domain id; the result only contains entries whose domain id starts with *filterDomain*. Inactive participants hosting the party are not shown in the result.

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`: Int

Return type:

- Seq[com.digitalasset.canton.admin.api.client.data.ListPartiesResult]

Description: This command allows you to deeply inspect the topology state used for synchronisation. The response is built from the timestamped topology transactions of each domain. The filterDomain parameter is used to filter the results by domain id; the result only contains entries whose domain id starts with *filterDomain*.

parties.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.list_by_owner

Summary: List keys for given keyOwner.

Arguments:

- `keyOwner`: com.digitalasset.canton.topology.KeyOwner
- `filterDomain`: String
- `asOf`: Option[java.time.Instant]
- `limit`: Int

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`: Int

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', 'DomainIdentityManagerId.Code'.

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.download

Summary: Download public key

Arguments:

- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint
- `outputFile`: Option[String]

Return type:

- com.digitalasset.canton.crypto.PublicKeyWithName

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:

- `key`: com.digitalasset.canton.crypto.PublicKey

- `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.public.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

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
- `outputFile`: Option[String]

Return type:

- com.digitalasset.canton.crypto.v0.CryptoKeyPair

keys.secret.upload

Summary: Upload a key pair

Arguments:

- `pair`: com.digitalasset.canton.crypto.v0.CryptoKeyPair
- `name`: Option[String]

keys.secret.upload

Summary: Upload (load and import) a key pair from file

Arguments:

- `filename`: String
- `name`: Option[String]

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.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.PublicKeyWithName]

Description: Returns all public keys to the corresponding private keys in the key vault. Optional arguments can be used for filtering.

keys.secret.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

certs.load (Preview)

Summary: Import X509 certificate in PEM format

Arguments:

- `x509Pem`: String

Return type:

- String

certs.list (Preview)

Summary: List locally stored certificates

Arguments:

- `filterUid`: String

Return type:

- List[com.digitalasset.canton.admin.api.client.data.CertificateResult]

certs.generate (Preview)

Summary: Generate a self-signed certificate

Arguments:

- `uid`: com.digitalasset.canton.topology.UniqueIdentifier
- `certificateKey`: com.digitalasset.canton.crypto.Fingerprint
- `additionalSubject`: String
- `subjectAlternativeNames`: Seq[String]

Return type:

- com.digitalasset.canton.admin.api.client.data.CertificateResult

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.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.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 from a user-chosen string and a fingerprint of a signing key. The signing key is the root key of said namespace. 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 preseve 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.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.stores.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

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.

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.TimeoutDuration]

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. 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.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

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.

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.TimeoutDuration]

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 fin-

gerprint 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.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

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)

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.TimeoutDuration]
- `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, DomainIdentityManager) 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.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

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).

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.TimeoutDuration]
- `replaceExisting`: 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.participant_domain_states.active

Summary: Returns true if the given participant is currently active on the given domain

Arguments:

- `domainId`: com.digitalasset.canton.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.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.TimeoutDuration]
- `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.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

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. Time-

Query.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.

topology.participant_domain_states.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

topology.legal_identities.authorize (Preview)

Summary: Authorize a legal identity claim transaction

Arguments:

- `ops`: com.digitalasset.canton.topology.transaction.TopologyChangeOp
- `claim`: com.digitalasset.canton.topology.transaction.SignedLegalIdentityClaim
- `signedBy`: Option[com.digitalasset.canton.crypto.Fingerprint]
- `synchronize`: Option[com.digitalasset.canton.config.TimeoutDuration]

Return type:

- com.google.protobuf.ByteString

topology.legal_identities.generate_x509 (Preview)

Summary: Generate a signed legal identity claim for a specific X509 certificate

Arguments:

- `uid`: com.digitalasset.canton.topology.UniqueIdentifier
- `certificateId`: com.digitalasset.canton.crypto.CertificateId

Return type:

- com.digitalasset.canton.topology.transaction.SignedLegalIdentityClaim

topology.legal_identities.generate (Preview)

Summary: Generate a signed legal identity claim

Arguments:

- `claim`: com.digitalasset.canton.topology.transaction.LegalIdentityClaim

Return type:

- com.digitalasset.canton.topology.transaction.SignedLegalIdentityClaim

topology.legal_identities.list_x509 (Preview)

Summary: List legal identities with X509 certificates

Arguments:

- `filterStore`: String
- `useStateStore`: Boolean
- `timeQuery`: com.digitalasset.canton.topology.store.TimeQuery
- `operation`: Option[com.digitalasset.canton.topology.transaction.TopologyChangeOp]
- `filterUid`: String
- `filterSigningKey`: String

Return type:

- Seq[(com.digitalasset.canton.topology.UniqueIdentifier, com.digitalasset.canton.crypto.X509Certificate)]

Description: List the X509 certificates used as legal identities associated with a unique identifier. A legal identity allows to establish a link between an unique identifier and some external evidence of legal identity. Currently, the only X509 certificate are supported as evidence. Except for the CCF integration that requires participants to possess a valid X509

certificate, legal identities have no functional use within the system. They are purely informational. 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.

topology.legal_identities.list (Preview)

Summary: List legal identities

Arguments:

- `filterStore`: String
- `useStateStore`: Boolean
- `timeQuery`: com.digitalasset.canton.topology.store.TimeQuery
- `operation`: Option[com.digitalasset.canton.topology.transaction.TopologyChangeOp]
- `filterUid`: String
- `filterSigningKey`: String

Return type:

- Seq[com.digitalasset.canton.admin.api.client.data.ListSignedLegalIdentityClaimResult]

Description: List the legal identities associated with a unique identifier. A legal identity allows to establish a link between an unique identifier and some external evidence of legal identity. Currently, the only type of evidence supported are X509 certificates. Except for the CCF integration that requires participants to possess a valid X509 certificate, legal identities have no functional use within the system. They are purely informational. 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.

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

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.

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.TimeoutDuration]
- `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.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]

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 .

topology.all.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

Ledger API Access

The following commands on a participant reference provide access to the participant's Ledger API services.

ledger_api.ledger_id (Testing)

Summary: Get ledger id

Return type:

- String

ledger_api.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

Transaction Service

ledger_api.transactions.domain_of (Testing)

Summary: Get the domain that a transaction was committed over.

Arguments:

- `transactionId`: String

Return type:

- com.digitalasset.canton.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.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.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.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.TimeoutDuration

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.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.TimeoutDuration

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.

ledger_api.transactions.end (Testing)

Summary: Get ledger end

Return type:

- com.daml.ledger.api.v1.ledger_offset.LedgerOffset

ledger_api.transactions.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

Command Service

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.daml.ledger.api.DeduplicationPeriod]
- `submissionId`: String
- `minLedgerTimeAbs`: Option[java.time.Instant]

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.TimeoutDuration]
- `deduplicationPeriod`: Option[com.daml.ledger.api.DeduplicationPeriod]
- `submissionId`: String
- `minLedgerTimeAbs`: Option[java.time.Instant]

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.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.TimeoutDuration]
- deduplicationPeriod: Option[com.daml.ledger.api.DeduplicationPeriod]
- submissionId: String
- minLedgerTimeAbs: Option[java.time.Instant]

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.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- methodName: String

Command Completion Service

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.TimeoutDuration
- 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.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.TimeoutDuration
- 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.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

Active Contract Service

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.TimeoutDuration

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.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.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.TimeoutDuration

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.TimeoutDuration): 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.TimeoutDuration

Description: Will throw an exception if the contract is not found to be active within the given timeout

ledger_api.acs.of_all (Testing)

Summary: List the set of active contracts for all parties hosted on this participant

Arguments:

- `limit`: Option[Int]
- `verbose`: Boolean
- `filterTemplates`: Seq[com.daml.ledger.client.binding.Primitive.TemplateId[_]]

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`: Option[Int]
- `verbose`: Boolean
- `filterTemplates`: Seq[com.daml.ledger.client.binding.Primitive.TemplateId[_]]

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.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

Package Service

ledger_api.packages.list (Testing)

Summary: List Daml Packages

Arguments:

- `limit`: Option[Int]

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.

ledger_api.packages.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

Party Management Service

ledger_api.parties.list (Testing)

Summary: List parties known by the ledger API server

Return type:

- Seq[com.daml.ledger.api.v1.admin.party_management_service.PartyDetails]

ledger_api.parties.allocate (Testing)

Summary: Allocate new party

Arguments:

- `party`: String
- `displayName`: String

Return type:

- com.daml.ledger.api.v1.admin.party_management_service.PartyDetails

ledger_api.parties.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

Ledger Configuration Service

ledger_api.configuration.list (Testing)

Summary: Obtain the ledger configuration
Arguments:
- `expectedConfigs`: Int
- `timeout`: com.digitalasset.canton.config.TimeoutDuration

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.configuration.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

Ledger Api User Management Service

ledger_api.users.list (Testing)

Summary: List users
Arguments:
- `filterUser`: String
- `pageToken`: String
- `pageSize`: Int

Return type:
- com.digitalasset.canton.admin.api.client.data.ListLedgerApiUsersResult

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

ledger_api.users.delete (Testing)

Summary: Delete user
Arguments:
- `id`: String

Description: Delete a user.

ledger_api.users.create (Testing)

Summary: Create a user with the given id
Arguments:
- `id`: String
- `actAs`: Set[com.digitalasset.canton.LfPartyId]
- `primaryParty`: Option[com.digitalasset.canton.LfPartyId]
- `readAs`: Set[com.digitalasset.canton.LfPartyId]
- `participantAdmin`: Boolean

Return type:
- com.digitalasset.canton.admin.api.client.data.LedgerApiUser

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

ledger_api.users.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

Return type:

- com.digitalasset.canton.admin.api.client.data.UserRights

Description: Lists the rights of a user, or the rights of the current user.

ledger_api.users.rights.revoke (Testing)

Summary: Revoke user rights

Arguments:

- `id`: String
- `actAs`: Set[com.digitalasset.canton.LfPartyId]
- `readAs`: Set[com.digitalasset.canton.LfPartyId]
- `participantAdmin`: Boolean

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

ledger_api.users.rights.grant (Testing)

Summary: Grant new rights to a user

Arguments:

- `id`: String
- `actAs`: Set[com.digitalasset.canton.LfPartyId]
- `readAs`: Set[com.digitalasset.canton.LfPartyId]
- `participantAdmin`: Boolean

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

ledger_api.users.rights.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

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:

- com.digitalasset.canton.admin.api.client.data.LedgerMeteringReport

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.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.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
- `originDomain`: 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.search (Preview)

Summary: Search the currently in-flight transfers

Arguments:

- `targetDomain`: com.digitalasset.canton.DomainAlias
- `filterOriginDomain`: Option[com.digitalasset.canton.DomainAlias]
- `filterTimestamp`: Option[java.time.Instant]
- `filterSubmittingParty`: Option[com.digitalasset.canton.topology.PartyId]
- `limit`: Int

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.

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

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.

transfer.out (Preview)

Summary: Transfer-out a contract from the origin domain with destination target domain

Arguments:

- `submittingParty`: com.digitalasset.canton.topology.PartyId
- `contractId`: com.digitalasset.canton.protocol.LfContractId
- `originDomain`: com.digitalasset.canton.DomainAlias
- `targetDomain`: com.digitalasset.canton.DomainAlias

Return type:

- com.digitalasset.canton.protocol.TransferId

Description: Transfers the given contract out of the origin 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 origin domain. It must also be connected to the target domain.

transfer.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

Ledger 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.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.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.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.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.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

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.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]]

commitments.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

Participant Repair

repair.unignore_events

Summary: Remove the ignored status from sequenced events.

Arguments:

- `domainId`: com.digitalasset.canton.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.ignore_events

Summary: Mark sequenced events as ignored.

Arguments:

- `domainId`: com.digitalasset.canton.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.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

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.

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 possi-

ble 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.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

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.

repair.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

Resource Management

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 two kinds of limits: *max_dirty_requests* and *max_rate*. 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 max-

imum 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. 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, even when it sends a burst of commands. To determine a suitable value for *max_dirty_requests*, you should test the system under high load. If you choose a higher value, throughput may increase, as more commands are validated in parallel. If you observe a high latency (time between submission and observing a command completion) or even command timeouts, you should choose a lower value. Once a suitable value for *max_dirty_requests* has been found, you should include bursts into the tests to also find a suitable value for *max_rate*. 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.

resources.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

Replication

replication.set_passive

Summary: Set the participant replica to passive

Description: Trigger a graceful fail-over from this active replica to another passive replica.

replication.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

3.3.6.8 Domain Administration Commands

config

Summary: Returns the domain configuration

Return type:

- LocalDomainReference.this.consoleEnvironment.environment.config.DomainConfigType

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

stop

Summary: Stop the instance

start

Summary: Start the instance

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.DomainId

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.

help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

Health

health.wait_for_initialized

Summary: Wait for the node to be initialized

health.wait_for_running

Summary: Wait for the node to be running

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.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

Database

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.

db.migrate

Summary: Migrates the instance's database if using a database storage

db.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

Participants

participants.active

Summary: Test whether a participant is permissioned on this domain

Arguments:

- `participantId`: com.digitalasset.canton.topology.ParticipantId

Return type:

- Boolean

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.TimeoutDuration]

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

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.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

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.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.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

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.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.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.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.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

sequencer.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

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.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.initialize

Summary: Initialize a mediator

Arguments:

- `domainId`: com.digitalasset.canton.DomainId
- `mediatorId`: com.digitalasset.canton.topology.MediatorId
- `domainParameters`: com.digitalasset.canton.protocol.StaticDomainParameters
- `sequencerConnection`: com.digitalasset.canton.sequencing.SequencerConnection
- `topologySnapshot`: Option[com.digitalasset.canton.topology.store.StoredTopologyTransactions[com.digitalasset.canton.topology.transaction.TopologyChangeOp.Positive]]
- `cryptoType`: String

Return type:

- com.digitalasset.canton.crypto.PublicKey

mediator.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

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.TimeoutDuration

mediator.testing.fetch_domain_time (Testing)

Summary: Fetch the current time from the domain

Arguments:

- `timeout`: com.digitalasset.canton.config.TimeoutDuration

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.list_by_owner

Summary: List keys for given keyOwner.

Arguments:

- `keyOwner`: com.digitalasset.canton.topology.KeyOwner
- `filterDomain`: String
- `asOf`: Option[java.time.Instant]
- `limit`: Int

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`: Int

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', 'DomainIdentityManagerId.Code'.

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.download

Summary: Download public key

Arguments:

- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint
- `outputFile`: Option[String]

Return type:

- com.digitalasset.canton.crypto.PublicKeyWithName

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:

- `key`: com.digitalasset.canton.crypto.PublicKey
- `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.public.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

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
- `outputFile`: Option[String]

Return type:

- com.digitalasset.canton.crypto.v0.CryptoKeyPair

keys.secret.upload

Summary: Upload a key pair

Arguments:

- `pair`: com.digitalasset.canton.crypto.v0.CryptoKeyPair
- `name`: Option[String]

keys.secret.upload

Summary: Upload (load and import) a key pair from file

Arguments:

- `filename`: String
- `name`: Option[String]

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.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.PublicKeyWithName]

Description: Returns all public keys to the corresponding private keys in the key vault. Optional arguments can be used for filtering.

keys.secret.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

certs.load (Preview)

Summary: Import X509 certificate in PEM format

Arguments:

- `x509Pem`: String

Return type:

- String

certs.list (Preview)

Summary: List locally stored certificates

Arguments:

- `filterUid`: String

Return type:

- List[com.digitalasset.canton.admin.api.client.data.CertificateResult]

certs.generate (Preview)

Summary: Generate a self-signed certificate

Arguments:

- `uid`: com.digitalasset.canton.topology.UniqueIdentifier
- `certificateKey`: com.digitalasset.canton.crypto.Fingerprint
- `additionalSubject`: String
- `subjectAlternativeNames`: Seq[String]

Return type:

- com.digitalasset.canton.admin.api.client.data.CertificateResult

Parties

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`: Int

Return type:

- Seq[com.digitalasset.canton.admin.api.client.data.ListPartiesResult]

Description: This command allows you to deeply inspect the topology state used for synchronisation. The response is built from the timestamped topology transactions of each domain. The filterDomain parameter is used to filter the results by domain id; the result only contains entries whose domain id starts with *filterDomain*.

parties.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

Service

service.update_dynamic_parameters

Summary: Update the Dynamic Domain Parameters for the domain

Arguments:

- `modifier`: com.digitalasset.canton.protocol.DynamicDomainParameters => com.digitalasset.canton.protocol.DynamicDomainParameters

service.set_dynamic_domain_parameters

Summary: Set the Dynamic Domain Parameters configured for the domain

Arguments:
- `dynamicDomainParameters`: com.digitalasset.canton.protocol.DynamicDomainParameters

service.get_dynamic_domain_parameters

Summary: Get the Dynamic Domain Parameters configured for the domain
Return type:
- com.digitalasset.canton.protocol.DynamicDomainParameters

service.get_static_domain_parameters

Summary: Get the Static Domain Parameters configured for the domain
Return type:
- com.digitalasset.canton.protocol.StaticDomainParameters

service.list_accepted_agreements

Summary: List the accepted service agreements
Return type:
- Seq[com.digitalasset.canton.domain.service.ServiceAgreementAcceptance]

service.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

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.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.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 from a user-chosen string and a fingerprint of a signing key. The signing key is the root key of said namespace. 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 preseve 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.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.stores.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

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.

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.TimeoutDuration]

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. 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.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

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.

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.TimeoutDuration]

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.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

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)

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.TimeoutDuration]
- `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, DomainIdentityManager) 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.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

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).

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.TimeoutDuration]
- `replaceExisting`: 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.participant_domain_states.active

Summary: Returns true if the given participant is currently active on the given domain

Arguments:

- `domainId`: com.digitalasset.canton.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.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.TimeoutDuration]
- `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.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

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.

topology.participant_domain_states.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:
- `methodName`: String

topology.legal_identities.authorize (Preview)

Summary: Authorize a legal identity claim transaction

Arguments:
- `ops`: com.digitalasset.canton.topology.transaction.TopologyChangeOp
- `claim`: com.digitalasset.canton.topology.transaction.SignedLegalIdentityClaim
- `signedBy`: Option[com.digitalasset.canton.crypto.Fingerprint]
- `synchronize`: Option[com.digitalasset.canton.config.TimeoutDuration]

Return type:
- com.google.protobuf.ByteString

topology.legal_identities.generate_x509 (Preview)

Summary: Generate a signed legal identity claim for a specific X509 certificate

Arguments:
- `uid`: com.digitalasset.canton.topology.UniqueIdentifier
- `certificateId`: com.digitalasset.canton.crypto.CertificateId

Return type:
- com.digitalasset.canton.topology.transaction.SignedLegalIdentityClaim

topology.legal_identities.generate (Preview)

Summary: Generate a signed legal identity claim

Arguments:

- `claim`: com.digitalasset.canton.topology.transaction.LegalIdentityClaim

Return type:

- com.digitalasset.canton.topology.transaction.SignedLegalIdentityClaim

topology.legal_identities.list_x509 (Preview)

Summary: List legal identities with X509 certificates

Arguments:

- `filterStore`: String
- `useStateStore`: Boolean
- `timeQuery`: com.digitalasset.canton.topology.store.TimeQuery
- `operation`: Option[com.digitalasset.canton.topology.transaction.TopologyChangeOp]
- `filterUid`: String
- `filterSigningKey`: String

Return type:

- Seq[(com.digitalasset.canton.topology.UniqueIdentifier, com.digitalasset.canton.crypto.X509Certificate)]

Description: List the X509 certificates used as legal identities associated with a unique identifier. A legal identity allows to establish a link between an unique identifier and some external evidence of legal identity. Currently, the only X509 certificate are supported as evidence. Except for the CCF integration that requires participants to possess a valid X509 certificate, legal identities have no functional use within the system. They are purely informational. 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.

topology.legal_identities.list (Preview)

Summary: List legal identities

Arguments:

- `filterStore`: String
- `useStateStore`: Boolean
- `timeQuery`: com.digitalasset.canton.topology.store.TimeQuery
- `operation`: Option[com.digitalasset.canton.topology.transaction.TopologyChangeOp]
- `filterUid`: String
- `filterSigningKey`: String

Return type:

- Seq[com.digitalasset.canton.admin.api.client.data.ListSignedLegalIdentityClaimResult]

Description: List the legal identities associated with a unique identifier. A legal identity allows to establish a link between an unique identifier and some external evidence of legal identity. Currently, the only type of evidence supported are X509 certificates. Ex-

cept for the CCF integration that requires participants to possess a valid X509 certificate, legal identities have no functional use within the system. They are purely informational. 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.

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

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.

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.TimeoutDuration]
- `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.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]

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 .

topology.all.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

3.3.6.9 Domain Manager Administration Commands

config

Summary: Returns the domain configuration
Return type:
- com.digitalasset.canton.domain.config.EnterpriseDomainManagerConfig

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

stop

Summary: Stop the instance

start

Summary: Start the instance

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.DomainId

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.

help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

Setup

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.

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.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.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

Health

health.wait_for_initialized

Summary: Wait for the node to be initialized

health.wait_for_running

Summary: Wait for the node to be running

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.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

Database

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.

db.migrate

Summary: Migrates the instance's database if using a database storage

db.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.list_by_owner

Summary: List keys for given keyOwner.
Arguments:
- `keyOwner`: com.digitalasset.canton.topology.KeyOwner
- `filterDomain`: String
- `asOf`: Option[java.time.Instant]
- `limit`: Int

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`: Int

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', 'DomainIdentityManagerId.Code'.

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.download

Summary: Download public key

Arguments:

- `fingerprint`: com.digitalasset.canton.crypto.Fingerprint
- `outputFile`: Option[String]

Return type:

- com.digitalasset.canton.crypto.PublicKeyWithName

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:

- `key`: com.digitalasset.canton.crypto.PublicKey
- `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.public.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

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
- `outputFile`: Option[String]

Return type:

- com.digitalasset.canton.crypto.v0.CryptoKeyPair

keys.secret.upload

Summary: Upload a key pair

Arguments:

- `pair`: com.digitalasset.canton.crypto.v0.CryptoKeyPair
- `name`: Option[String]

keys.secret.upload

Summary: Upload (load and import) a key pair from file

Arguments:

- `filename`: String
- `name`: Option[String]

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.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.PublicKeyWithName]

Description: Returns all public keys to the corresponding private keys in the key vault. Optional arguments can be used for filtering.

keys.secret.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

certs.load (Preview)

Summary: Import X509 certificate in PEM format
Arguments:
- `x509Pem`: String

Return type:
- String

certs.list (Preview)

Summary: List locally stored certificates
Arguments:
- `filterUid`: String

Return type:
- List[com.digitalasset.canton.admin.api.client.data.CertificateResult]

certs.generate (Preview)

Summary: Generate a self-signed certificate
Arguments:
- `uid`: com.digitalasset.canton.topology.UniqueIdentifier
- `certificateKey`: com.digitalasset.canton.crypto.Fingerprint
- `additionalSubject`: String
- `subjectAlternativeNames`: Seq[String]

Return type:
- com.digitalasset.canton.admin.api.client.data.CertificateResult

Parties

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`: Int

Return type:

- Seq[com.digitalasset.canton.admin.api.client.data.ListPartiesResult]

Description: This command allows you to deeply inspect the topology state used for synchronisation. The response is built from the timestamped topology transactions of each domain. The filterDomain parameter is used to filter the results by domain id; the result only contains entries whose domain id starts with *filterDomain*.

parties.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

Service

service.update_dynamic_parameters

Summary: Update the Dynamic Domain Parameters for the domain

Arguments:

- `modifier`: com.digitalasset.canton.protocol.DynamicDomainParameters => com.digitalasset.canton.protocol.DynamicDomainParameters

service.set_dynamic_domain_parameters

Summary: Set the Dynamic Domain Parameters configured for the domain

Arguments:

- `dynamicDomainParameters`: com.digitalasset.canton.protocol.DynamicDomainParameters

service.get_dynamic_domain_parameters

Summary: Get the Dynamic Domain Parameters configured for the domain

Return type:

- com.digitalasset.canton.protocol.DynamicDomainParameters

service.get_static_domain_parameters

Summary: Get the Static Domain Parameters configured for the domain

Return type:

- com.digitalasset.canton.protocol.StaticDomainParameters

service.list_accepted_agreements

Summary: List the accepted service agreements

Return type:

- Seq[com.digitalasset.canton.domain.service.ServiceAgreementAcceptance]

service.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

Topology Administration

Same as *Domain Topology Administration*.

3.3.6.10 Sequencer Administration Commands

ethereum.deploy_sequencer_contract

Summary:

This function attempts to deploy the Solidity sequencer smart contract to the configured (E

On success, it returns the contract address and block height of the deployed sequencer contract.

Return type:

- (String, java.math.BigInteger)

Description: This function attempts to deploy the Solidity sequencer smart contract to the configured (Besu) network. On success, it returns the contract address and block height of the deployed sequencer contract. Note that this function can't be run over gRPC but needs to be used in a local Canton console. This function can only be executed when using an Ethereum sequencer.

ethereum.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

config

Summary: Returns the sequencer configuration

Return type:

- com.digitalasset.canton.domain.sequencing.SequencerNodeConfig

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

stop

Summary: Stop the instance

start

Summary: Start the instance

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

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.

help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

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.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.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

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.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.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.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.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

sequencer.help

Summary: Help for specific commands (use help() or help(method) for more informa-

tion)
Arguments:
- `methodName`: String

Health

health.wait_for_initialized

Summary: Wait for the node to be initialized

health.wait_for_running

Summary: Wait for the node to be running

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.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

Database

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.

db.migrate

Summary: Migrates the instance's database if using a database storage

db.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

3.3.6.11 Mediator Administration Commands

config

Summary: Returns the mediator configuration
Return type:
- com.digitalasset.canton.domain.mediator.MediatorNodeConfig

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

stop

Summary: Stop the instance

start

Summary: Start the instance

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

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.

help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

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.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.initialize

Summary: Initialize a mediator

Arguments:

- `domainId`: com.digitalasset.canton.DomainId
- `mediatorId`: com.digitalasset.canton.topology.MediatorId
- `domainParameters`: com.digitalasset.canton.protocol.StaticDomainParameters
- `sequencerConnection`: com.digitalasset.canton.sequencing.SequencerConnection
- `topologySnapshot`: Option[com.digitalasset.canton.topology.store.StoredTopologyTransactions[com.digitalasset.canton.topology.transaction.TopologyChangeOp.Positive]]
- `cryptoType`: String

Return type:

- com.digitalasset.canton.crypto.PublicKey

mediator.help

Summary: Help for specific commands (use help() or help(method) for more information)

Arguments:

- `methodName`: String

Health

health.wait_for_initialized

Summary: Wait for the node to be initialized

health.wait_for_running

Summary: Wait for the node to be running

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.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

Database

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.

db.migrate

Summary: Migrates the instance's database if using a database storage

db.help

Summary: Help for specific commands (use help() or help(method) for more information)
Arguments:
- `methodName`: String

3.3.6.12 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 ...
```

3.3.7 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 can not 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*.

3.3.7.1 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 can not 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.

3.3.7.2 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
            domain-parameters.unique-contract-keys = false
        }
    }
    participants {
        participant1 {
```

(continues on next page)

(continued from previous page)

```
            // subsequent changes have no effect and the mode of a node can never
↪be changed
            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"
  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

    controller sig can
      nonconsuming Initialize: Optional (ContractId Initialization)
        with
          k: Int
        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
  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
```

(continues on next page)

(continued from previous page)

```
    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

    controller sig can
      Generate : (ContractId Generator, ContractId Keyed)
        with
          k: Int
        do
          existing <- lookupByKey @Keyed (sig, k)
          keyed <- case existing of
            Some cid -> pure cid
            None ->
```

(continues on next page)

(continued from previous page)

```
            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

    controller sig can
      Allocate : (ContractId KeyState, ContractId Keyed)
        do
          assert $ not allocated
          newState <- create this with allocated = True
          keyed <- create Keyed with ..
          pure (newState, keyed)

      Deallocate : ContractId KeyState
        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
  -- KeyState, you must also recreate it in the same transaction.
```

(continues on next page)

(continued from previous page)

```
-- 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.

Important: This feature is only available in Canton Enterprise

3.3.8 Enterprise Drivers

The Canton Enterprise edition provides the following drivers in addition to the PostgreSQL-based domain in the Canton Community edition.

Important: This feature is only available in Canton Enterprise

3.3.8.1 Fabric Domain

The Canton-on-Fabric integration runs a Canton domain where events are sequenced using the Hyperledger Fabric ledger.

The Demo Deployment

To run the demo Canton Fabric deployment, you will need access to the following:

> a Canton Enterprise release for the example files and the Canton enterprise binary
> *Canton Enterprise docker repository* access, in order to have access to the docker images

Also make sure to have docker and docker-compose installed.

Basic Demo

The demo can be found inside 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 single`.

The script will start the following:

1. A Fabric ledger with 2 peers and one orderer node.
2. A Canton Sequencer node that interacts with the Fabric ledger.
3. A Canton process running a single Canton domain 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
```

(continues on next page)

(continued from previous page)

```
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
```

Troubleshooting

Some Mac users may report docker issues during the chaincode installation steps, which has been linked to this issue and can be easily fixed by doing the following:

Preferences -> Experimental Features -> Turn off Use gRPC FUSE for file sharing

Run the Example Deployment

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. There you will find config files, docker-compose files and scripts to set up all of or a subset of the following:

a Fabric ledger with 2 organizations (with one peer each) and one orderer node
a Fabric block explorer UI
one Canton domain and 2 participants
2 sequencers backed by the Fabric ledger

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. However, if you wish to use a different version, you can specify it with the CANTON_VERSION environment variable. For example, `export CANTON_VERSION=0.19.0` to use Canton v0.19.0. 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-sequencer-1.yaml`: Runs a canton-enterprise node with a Fabric sequencer in it backed by a Postgres cache.
`docker-compose-sequencer-2.yaml`: Runs a second Fabric sequencer inside the same canton node. This shows that multiple sequencers operating on top of the same Fabric ledgers are supported.
`docker-compose-canton.yaml`: Runs a canton domain and 2 participants using the Fabric sequencer(s).

Single Sequencer

To run Canton with a single Fabric Sequencer, run `./run.sh single`.

If you just want to run the sequencer, without starting Canton, you can run the same command with the `-only-sequencer` flag. Alternatively, you can run 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-sequencer-1.yaml up
```

The Fabric Sequencer will be available on `localhost:3000`.

Multiple Sequencers

To run Canton with 2 Fabric Sequencers in a multi-sequencer setup, run `./run.sh multiple`. If you just want to run the sequencers, without starting Canton, you can run the same command with the `-only-sequencer` flag. Alternatively, you can run 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-sequencer-1.yaml -f docker-compose-sequencer-2.yaml up
```

This will start 2 sequencers, one on `localhost:3000` and the second one on `localhost:3001`.

Blockchain Explorer

If you wish to start the Hyperledger Fabric Blockchain Explorer, you can do so by running one of the `./run.sh` commands shown above but also adding the `-e` flag. Alternatively you can use one of the docker-compose commands and add `-f docker-compose-blockchain-explorer.yaml` flag to them. You will then be able to see the explorer web UI in your browser if you go to `http://localhost:8080`.

Alternatively, 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
```

Note that even when the explorer is working perfectly, it might output some error messages like the following that can be safely ignored:

```
[ERROR] FabricGateway - Failed to get block 0 from channel undefined :
↪TypeError: Cannot read property 'toString' of undefined
```

Cleanup

When you're done running the sequencer, make sure to run `./run.sh down`.

Using the Canton Binary instead of docker

If you wish to run the domain and participants separately (with the canton binary or jar), you can do so as follows:

Single sequencer:

```
<<canton-release>>/examples/e01-fabric-domain/canton-on-fabric$ ./run.sh single -
↪only-sequencer

<<canton-release>>/examples/e01-fabric-domain/canton-on-fabric$ ../../../bin/
↪canton -c "config/canton/demo.conf" --bootstrap "config/canton/demo.canton"
```

Multiple sequencers:

```
<<canton-release>>/examples/e01-fabric-domain/canton-on-fabric$ ./run.sh multiple
↪-only-sequencer

<<canton-release>>/examples/e01-fabric-domain/canton-on-fabric$ ../../../bin/
↪canton -c "config/canton/demo.conf" --bootstrap "config/canton/demo.canton"
```

The `-only-sequencer` flag makes sure that the script does not start the domain and participants, allowing us to run it ourselves with the canton binary or the jar.

If you wish to run the jar file instead of the canton binary, simply replace `../../../bin/canton` above with `java -jar ../../../lib/canton-enterprise-*.jar`.

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 work, all it needs is access to a Fabric Ledger that has been setup with at least one channel where the Canton Sequencer chaincode has been installed.

In order to configure access to the ledger, please follow the notes and example show in `examples/e01-fabric-domain/canton-on-fabric/config/fabric/fabric-config-1.conf`. You'll need to provide the necessary urls and files as described in the sample config.

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`.

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).

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.

Important: This feature is only available in Canton Enterprise

---

3.3.8.2 Ethereum Domain

The Canton Enterprise Ethereum integration runs on an external Ethereum network. It is currently tested with Hyperledger Besu.

The Demo Deployment

To run the demo Canton Ethereum deployment, you will need access to a Canton Enterprise release, the *Canton Enterprise docker repository* , as well as having docker, docker-compose, and Hyperledger Besu (instructions here) installed.

Run the Demo Deployment

The demo Ethereum deployment can be found inside the examples directory of the Canton Enterprise release. Unpack the Canton Enterprise release and then `cd` into `examples/e03-ethereum-sequencer`.

The script `./ethereum-docker-compose/run.sh` from the folder `examples` will create a new Besu testnet for the demo deployment and then start the demo. In particular, the script will start several dockerised services:

> An ethereum testnet, using 4 Besu nodes with the IBFT consensus protocol.
> An instance of Canton. This includes two Participants and a Canton Enterprise Domain with two Ethereum sequencers. The configuration for the canton instance is in `ethereum-docker-compose/conf/config-example.conf`.

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 the Canton Enterprise release being used, but can alternatively be set to a different version or `dev` for the latest main build of Canton.

Start the demo deployment as follows:

```
<<canton-release>>/examples/e03-ethereum-sequencer$ CANTON_VERSION=<your version>
↪./ethereum-docker-compose/run.sh
```

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-release>>/examples/e03-ethereum-sequencer$ ../../bin/canton -c ethereum-
↪docker-compose/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-release>>/examples/e03-ethereum-sequencer$ ./stop-with-purge.sh
```

Generating a Clean Testnet

The directory `examples/e03-ethereum-sequencer/ibft-testnet` contains the script `generate-testnet.sh`. This automatically generates a clean Besu network, including creating new randomised private keys. This is automatically called by `run.sh` but you may want to understand and edit this script to create your own custom Besu deployment.

When `generate-testnet.sh` is run:

> A genesis file and a set of keys for four Besu nodes are automatically generated. These can be found in the folder `generated`.
> The four Besu nodes are started.
> The state from any previous runs of `generate-testnet.sh` is deleted.

The generated testnet includes an instance of Canton's Ethereum Sequencer contract pre-deployed from genesis. This contract has the address `0x0ffd23af8eebc60b3cfdeed6f814988757237314` and is defined by the binary file `sequencer-binary`. The original Solidity source code of the Sequencer smart contract is in `<<canton-release>>/drivers/ethereum/solidity/Sequencer.sol`.

The generated Besu testnet has been configured largely following these tutorials:

> https://besu.hyperledger.org/en/stable/Tutorials/Private-Network/Create-IBFT-Network/ and https://besu.hyperledger.org/en/stable/HowTo/Configure/FreeGas/

The parameters of the generated testnet can be changed by modifying the `genesis.json` file defined inline in `generate-testnet.sh`.

Customization of the Demo Configuration

You can also modify the Canton configuration and bootstrap script for the demo. The Canton configuration is found in `ethereum-docker-compose/conf/config-demo.conf` and the bootstrap script is `ethereum-docker-compose/ping.canton`. Note that if you change port mappings in the Canton config file then you may also need to update `ethereum-docker-compose/docker-compose.yaml`. The scaladoc documentation contains more details on the impact of specific Ethereum configuration options.

TLS configuration

Canton supports establishing a TLS connection to Ethereum client nodes. The demo showcases how to configure Canton to use mutual TLS authentication between Canton and one of the Besu nodes. In particular, the bash script `start-node.sh`, the mix-in Canton configuration `tls.conf` and the certificates in `ibft-testnet/tls` contain all settings pertaining to TLS. 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.

Configuring a wallet file

Canton allows configuring a wallet in UTC JSON and BIP 39 format. The Ethereum demo includes an example of a mix-in wallet configuration for a BIP39-based wallet in `ethereum-docker-compose/conf/wallet.conf`. A UTC JSON wallet configuration could look as follows:

```
canton.sequencers.ethereumSequencer1.sequencer.config.wallet {
  type = "utc-json-wallet"
  password = "supersafe"
  wallet-path = "./enterprise/app/src/test/resources/ethereum/wallet.json"
}
```

For more details, please refer to the Canton scaladoc documentation.

Deployment to an existing Ethereum Network

The demo deploys the Ethereum sequencer contract by passing it to the bootstrap node and deploying it in the genesis block. This isn't possible when attempting to use the Ethereum sequencer with an existing and already running network or blockchain. To deploy the Ethereum sequencer with an existing Ethereum network, we expose the utility console command deploy_sequencer_contract. Deployment to an existing network then works as follows:

1. Configure the Ethereum sequencer to be able to connect to the network but set the contract address and deployment height within the SequencerContractConfig to dummy values as these will be ignored for now. If you want to try these steps with the demo, you don't need to change anything in the configuration. 2. Create a bootstrap script that deploys the sequencer contract to the configured network via calling `deploy_sequencer_contract`. For example, for the demo we use script *sequencer_deployment.canton* with the following content:

```
val (address, block) = ethereumSequencer1.ethereum.deploy_sequencer_contract
println(s"Sequencer contract successfully deployed to contract address `$address`
↪in block `$block`")
```

It is important to specify `--manual-start` so the Canton nodes aren't started automatically when running the bootstrap script and Canton does not try to load the configured sequencer contract it expects at the address specified within the `SequencerContractConfig`.

3. Start Canton and execute the bootstrap script. For the demo, you will need to change *client-host* for *ethereumSequencer1* and *ethereumSequencer2* from respectively *besu-1* and *besu-2* to *127.0.0.1* in *conf/config-example.conf*. Then run the following command:

```
<<canton-release>>/bin/canton run ethereum-docker-compose/sequencer_deployment.
↪canton --config ethereum-docker-compose/conf/config-example.conf --manual-start
```

You should see output similar to the following:

```
Sequencer contract successfully deployed to contract address
↪`0x68f21d0613f2c9b1e677f09ba6a9eb0980d6d5c1` in block `506`
```

Stop Canton next (`exit`).

4. Update the contract address and deployment height in the `SequencerContractConfig` configuration (file `ethereum-docker-compose/conf/config-example.conf` for the demo) with the output from the previous step.

5. Start Canton normally, e.g., with the following command for the demo:

```
<<canton-release>>/bin/canton --config ethereum-docker-compose/conf/config-
↪example.conf --bootstrap ethereum-docker-compose/ping.canton
```

If everything went well, you should see the ping succeeding and the interactive console starting.

Requirements for the Ethereum Network

The Canton Ethereum integration is currently tested with a Hyperledger Besu free gas network, running with the IBFT 2.0 consensus protocol. However, the integration only interfaces with the Ethereum client via web3j so most popular Ethereum clients (eg Geth and Parity) should be supported. There are, however, some requirements for the Ethereum Network irrespective of the Ethereum client you choose to use:

> 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.

3.3.9 Error codes

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)**. These error codes allow a developer, user or operator to identify the exact error occurring such that an error can be automatically handled or looked up in the documentation.

In the above example, 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 to this request, or 0 if no correlation-id is given.

The purpose of the correlation-id is to allow a user to clearly identify the request, such that the operator can lookup any log information associated with this error.

The majority of the errors are a result of some request processing. Such errors are logged, with a log level usually depending on the category, and returned to the user as a failed gRPC request (using the standard StatusRuntimeException). In some 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 rather unusual behaviour.
ERROR to log internal errors within the system, 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.

3.3.9.1 Error Categories

The error categories allow 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.

The error categories are currently documented as part of our scaladoc reference page.

3.3.9.2 Machine Readable Information

Every error on the API is constructed in a way that allows automated and manual error handling. First, the error category will map to exactly one gRPC status code. Second, every error description (of the corresponding `StatusRuntimeException.Status`) will start with the error information (**SOMETHING_NOT_SO_GOOD_HAPPENED(CN,x)**), separated from a human readable description using a colon (:). The rest of the description is targeting 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`. Many errors will include more information, but there is no guarantee given 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. As an 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.

3.3.9.3 List of error codes

1. ParticipantErrorGroup

1.1. 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

1.1.1. MismatchError

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: BackgroundProcessDegradationWarning
Conveyance: This error is logged with log-level WARN on the server side.
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: BackgroundProcessDegradationWarning
Conveyance: This error is logged with log-level WARN on the server side.
Scaladocs: ACS_MISMATCH_NO_SHARED_CONTRACTS

1.2. LedgerApiErrors

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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: CONFIGURATION_ENTRY_REJECTED

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. This error is exposed on the API with grpc-status INTERNAL without any details due to security reasons
Scaladocs: LEDGER_API_INTERNAL_ERROR

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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: PACKAGE_UPLOAD_REJECTED

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 WARN on the server side. This error is exposed on the API with grpc-status ABORTED including a detailed error message
Scaladocs: PARTICIPANT_BACKPRESSURE

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. This error is exposed on the API with grpc-status DEADLINE_EXCEEDED including a detailed error message
Scaladocs: REQUEST_TIME_OUT

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. This error is exposed on the API with grpc-status UNAVAILABLE including a detailed error message
Scaladocs: SERVICE_NOT_RUNNING

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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: TOO_MANY_USER_RIGHTS

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. This error is exposed on the API with grpc-status UNIMPLEMENTED without any details due to security reasons
Scaladocs: UNSUPPORTED_OPERATION

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. This error is 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. This error is exposed on the API with grpc-status NOT_FOUND including a detailed error message
Scaladocs: USER_NOT_FOUND

1.2.1. 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. This error is exposed on the API with grpc-status ABORTED including a detailed error message
Scaladocs: FAILED_TO_DETERMINE_LEDGER_TIME

1.2.1.1. 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. This error is 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: MaliciousOrFaultyBehaviour
Conveyance: This error is logged with log-level WARN on the server side. This error is exposed on the API with grpc-status UNKNOWN without any details due to security reasons
Scaladocs: PACKAGE_VALIDATION_FAILED

1.2.1.2. 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. This error is exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message
Scaladocs: COMMAND_PREPROCESSING_FAILED

1.2.1.3. Interpreter

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. This error is exposed on the API with grpc-status NOT_FOUND including a detailed error message
Scaladocs: CONTRACT_NOT_ACTIVE

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. This error is 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. This error is 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. This error is exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message
Scaladocs: DAML_INTERPRETER_INVALID_ARGUMENT

1.2.1.3.1. 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. This error is exposed on the API with grpc-status NOT_FOUND including a detailed error message
Scaladocs: CONTRACT_KEY_NOT_FOUND

1.2.2. 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. This error is exposed on the API with grpc-status NOT_FOUND including a detailed error message
Scaladocs: CONTRACT_NOT_FOUND

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. This error is 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. This error is 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. This error is 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. This error is 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. This error is 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. This error is 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. This error is exposed on the API with grpc-status ABORTED including a detailed error message
Scaladocs: SUBMISSION_ALREADY_IN_FLIGHT

1.2.3. 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. This error is 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. This error is 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. This error is exposed on the API with grpc-status INTERNAL without any details due to security reasons
Scaladocs: PACKAGE_SERVICE_INTERNAL_ERROR

1.2.3.1. 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. This error is 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. This error is 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. This error is 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. This error is 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. This error is 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. This error is exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message
Scaladocs: ZIP_BOMB

1.2.4. 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. This error is exposed on the API with grpc-status INTERNAL without any details due to 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. This error is 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. This error is 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. This error is exposed on the API with grpc-status PERMISSION_DENIED without any details due to 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. This error is exposed on the API with grpc-status NOT_FOUND including a detailed error message
Scaladocs: SUBMITTING_PARTY_NOT_KNOWN_ON_LEDGER

1.2.4.1. 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. This error is exposed on the API with grpc-status INTERNAL without any details due to 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. This error is exposed on the API with grpc-status INTERNAL without any details due to security reasons
Scaladocs: INTERNALLY_INCONSISTENT_KEYS

1.2.5. 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. This error is exposed on the API with grpc-status INTERNAL without any details due to 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. This error is exposed on the API with grpc-status PERMISSION_DENIED without any details due to 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. This error is 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. This error is exposed on the API with grpc-status UNAUTHENTICATED without any details due to security reasons
Scaladocs: UNAUTHENTICATED

1.2.6. 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. This error is 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. This error is 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. This error is 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. This error is 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. This error is 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. This error is 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. This error is 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. This error is 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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: PARTICIPANT_PRUNED_DATA_ACCESSED

1.2.6.1. 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. This error is 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. This error is exposed on the API with grpc-status NOT_FOUND including a detailed error message
Scaladocs: PACKAGE_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. This error is exposed on the API with grpc-status NOT_FOUND including a detailed error message
Scaladocs: TRANSACTION_NOT_FOUND

1.3. TransactionErrorGroup

1.3.1. 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. This error is 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. This error is exposed on the API with grpc-status INTERNAL without any details due to security reasons
Scaladocs: ROUTING_INTERNAL_ERROR

1.3.1.1. 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. This error is 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. This error is 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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: NO_COMMON_DOMAIN

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. This error is exposed on the API with grpc-status NOT_FOUND including a detailed error message
Scaladocs: NO_DOMAIN_ON_WHICH_ALL_SUBMITTERS_CAN_SUBMIT

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. This error is 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. This error is 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. This error is exposed on the API with grpc-status NOT_FOUND including a detailed error message
Scaladocs: UNKNOWN_INFORMEES

1.3.1.2. MalformedInputErrors

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. This error is exposed on the API with grpc-status NOT_FOUND including a detailed error message
Scaladocs: INVALID_DOMAIN_ALIAS

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. This error is 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. This error is exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message
Scaladocs: INVALID_SUBMITTER

1.3.1.3. ConfigurationErrors

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. This error is 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 is correctly connected.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side. This error is exposed on the API with grpc-status NOT_FOUND including a detailed error message
Scaladocs: SUBMISSION_DOMAIN_NOT_READY

1.3.2. 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. This error is exposed on the API with grpc-status ABORTED including a detailed error message
Scaladocs: CHOSEN_MEDIATOR_IS_INACTIVE

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 WARN on the server side. This error is 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. This error is 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. This error is 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. This error is exposed on the API with grpc-status ABORTED including a detailed error message
Scaladocs: NOT_SEQUENCED_TIMEOUT

PACKAGE_NO_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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: PACKAGE_NO_VETTED_BY_RECIPIENTS

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. This error is 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. This error is 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. This error is exposed on the API with grpc-status ABORTED including a detailed error message
Scaladocs: SUBMISSION_DURING_SHUTDOWN

1.3.3. 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. This error is exposed on the API with grpc-status INTERNAL without any details due to 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. This error is 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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: NOT_CONNECTED_TO_ANY_DOMAIN

1.3.4. LocalReject

1.3.4.1. 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: MaliciousOrFaultyBehaviour
Conveyance: This error is logged with log-level WARN on the server side. This error is exposed on the API with grpc-status UNKNOWN without any details due to security reasons
Scaladocs: LOCAL_VERDICT_BAD_ROOT_HASH_MESSAGES

LOCAL_VERDICT_DETECTED_MULTIPLE_CONFIRMATION_POLICIES

Explanation: This rejection is made by a participant if a transaction uses different confirmation policies per view.
Resolution: This indicates either malicious or faulty behaviour.
Category: MaliciousOrFaultyBehaviour
Conveyance: This error is logged with log-level WARN on the server side. This error is exposed on the API with grpc-status UNKNOWN without any details due to security reasons
Scaladocs: LOCAL_VERDICT_DETECTED_MULTIPLE_CONFIRMATION_POLICIES

LOCAL_VERDICT_EMPTY_REJECTION

Explanation: This rejection is emitted by a participant if it receives an aggregated reject without any reason.
Resolution: This indicates either malicious or faulty mediator.
Category: MaliciousOrFaultyBehaviour
Conveyance: This error is logged with log-level WARN on the server side. This error is exposed on the API with grpc-status UNKNOWN without any details due to security reasons
Scaladocs: LOCAL_VERDICT_EMPTY_REJECTION

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: MaliciousOrFaultyBehaviour
Conveyance: This error is logged with log-level WARN on the server side. This error is exposed on the API with grpc-status UNKNOWN without any details due to 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: MaliciousOrFaultyBehaviour
Conveyance: This error is logged with log-level WARN on the server side. This error is exposed on the API with grpc-status UNKNOWN without any details due to security reasons
Scaladocs: LOCAL_VERDICT_MALFORMED_PAYLOAD

1.3.4.2. ConsistencyRejections

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: MaliciousOrFaultyBehaviour
Conveyance: This error is logged with log-level WARN on the server side. This error is exposed on the API with grpc-status UNKNOWN without any details due to security reasons
Scaladocs: LOCAL_VERDICT_CREATES_EXISTING_CONTRACTS

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. This error is 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. This error is 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. This error is 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. This error is 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. This error is exposed on the API with grpc-status ABORTED including a detailed error message
Scaladocs: LOCAL_VERDICT_LOCKED_KEYS

1.3.4.3. 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. For short-running transactions, simply retry.
Category: ContentionOnSharedResources
Conveyance: This error is logged with log-level INFO on the server side. This error is 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. This error is 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. This error is exposed on the API with grpc-status ABORTED including a detailed error message
Scaladocs: LOCAL_VERDICT_TIMEOUT

1.3.4.4. 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. This error is 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. This error is 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. This error is 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. This error is exposed on the API with grpc-status ABORTED including a detailed error message
Scaladocs: TRANSFER_IN_CONTRACT_IS_LOCKED

1.3.4.5. 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. This error is exposed on the API with grpc-status NOT_FOUND including a detailed error message
Scaladocs: TRANSFER_OUT_ACTIVENESS_CHECK_FAILED

1.3.5. 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. This error is exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message
Scaladocs: MALFORMED_DEDUPLICATION_OFFSET

1.3.6. MediatorReject

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 confirmation timeout window.
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. This error is exposed on the API with grpc-status ABORTED including a detailed error message
Scaladocs: MEDIATOR_SAYS_TX_TIMED_OUT

1.3.6.1. MaliciousSubmitter

MEDIATOR_SAYS_DECLARED_MEDIATOR_IS_WRONG

Explanation: This rejection indicates that the submitter sent the request to the wrong mediator
Resolution: Investigate whether the submitter is faulty or malicious.
Category: MaliciousOrFaultyBehaviour
Conveyance: This error is logged with log-level WARN on the server side. This error is exposed on the API with grpc-status UNKNOWN without any details due to security reasons
Scaladocs: MEDIATOR_SAYS_DECLARED_MEDIATOR_IS_WRONG

MEDIATOR_SAYS_NOT_ENOUGH_CONFIRMING_PARTIES

Explanation: This rejection indicates that a submitter has sent a manipulated view.
Resolution: Investigate whether the submitter is faulty or malicious.
Category: MaliciousOrFaultyBehaviour
Conveyance: This error is logged with log-level WARN on the server side. This error is exposed on the API with grpc-status UNKNOWN without any details due to security reasons
Scaladocs: MEDIATOR_SAYS_NOT_ENOUGH_CONFIRMING_PARTIES

MEDIATOR_SAYS_VIEW_THRESHOLD_BELOW_MINIMUM_THRESHOLD

Explanation: This rejection indicates that a submitter has sent a manipulated view.
Resolution: Investigate whether the submitter is faulty or malicious.
Category: MaliciousOrFaultyBehaviour
Conveyance: This error is logged with log-level WARN on the server side. This error is exposed on the API with grpc-status UNKNOWN without any details due to security reasons
Scaladocs: MEDIATOR_SAYS_VIEW_THRESHOLD_BELOW_MINIMUM_THRESHOLD

1.3.6.2. Topology

MEDIATOR_SAYS_INFORMEES_NOT_HOSTED_ON_ACTIVE_PARTICIPANTS

Explanation: The transaction is referring to informees that are not hosted on any active participant on this domain.
Resolution: This error can happen either if the transaction is racing with a topology state change, or due to malicious or faulty behaviour.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level WARN on the server side. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: MEDIATOR_SAYS_INFORMEES_NOT_HOSTED_ON_ACTIVE_PARTICIPANTS

MEDIATOR_SAYS_INVALID_ROOT_HASH_MESSAGES

Explanation: This rejection indicates that a submitter has sent a view with invalid root hash messages.
Resolution: This error can happen either if the transaction is racing with a topology state change, or due to malicious or faulty behaviour.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level WARN on the server side. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: MEDIATOR_SAYS_INVALID_ROOT_HASH_MESSAGES

1.4. SyncServiceError

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. This error is 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. This error is 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. This error is 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. This error is exposed on the API with grpc-status INTERNAL without any details due to security reasons
Scaladocs: SYNC_SERVICE_DOMAIN_DISCONNECTED

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. This error is exposed on the API with grpc-status INTERNAL without any details due to 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.
Category: InvalidGivenCurrentSystemStateResourceMissing
Conveyance: This error is logged with log-level INFO on the server side. This error is exposed on the API with grpc-status NOT_FOUND including a detailed error message
Scaladocs: SYNC_SERVICE_UNKNOWN_DOMAIN

1.4.1. 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. This error is exposed on the API with grpc-status INTERNAL without any details due to security reasons
Scaladocs: DOMAIN_REGISTRY_INTERNAL_ERROR

1.4.1.1. 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. This error is 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 INFO on the server side. This error is 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. This error is 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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: INVALID_DOMAIN_CONNECTION

1.4.1.2. 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. This error is 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. This error is 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. This error is 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. This error is 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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: SERVICE_AGREEMENT_ACCEPTANCE_FAILED

1.4.1.3. 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. This error is 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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: FAILED_TO_CONNECT_TO_SEQUENCER

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. This error is 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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: PARTICIPANT_IS_NOT_ACTIVE

1.5. 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

1.6. IndexErrors

1.6.1. 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. This error is exposed on the API with grpc-status INTERNAL without any details due to 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. This error is exposed on the API with grpc-status INTERNAL without any details due to 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. This error is exposed on the API with grpc-status UNAVAILABLE including a detailed error message
Scaladocs: INDEX_DB_SQL_TRANSIENT_ERROR

1.7. 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. This error is exposed on the API with grpc-status INTERNAL without any details due to 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. This error is 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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: PRUNING_NOT_SUPPORTED_IN_COMMUNITY_EDITION

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 your adjust the settings for this participant.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: UNSAFE_TO_PRUNE

1.8. CantonPackageServiceError

PACKAGE_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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: PACKAGE_REMOVAL_ERROR

1.9. 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. This error is exposed on the API with grpc-status INTERNAL without any details due to 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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: PARTICIPANT_REPLICATION_NOT_SUPPORTED_BY_STORAGE

2. EthereumErrors

2.1. 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. If this occurs, the sequencer ignores the configured or persisted *blockToReadFrom* and starts reading from the latest block mined by the blockchain (100 in this example).
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 (Console command *ethereum.deploy_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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: ATTEMPT_TO_CHANGE_IMMUTABLE_VALUE

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: Change the configuration of *blockToReadFrom* for the Ethereum sequencer when working with an existing (not fresh) Ethereum 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: 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. This error is 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 (see also *ethereum.deploy_sequencer_contract*).
Category: MaliciousOrFaultyBehaviour
Conveyance: This error is logged with log-level WARN on the server side. This error is exposed on the API with grpc-status UNKNOWN without any details due to security reasons
Scaladocs: WRONG_EVM_BYTECODE

3. TopologyManagementErrorGroup

3.1. TopologyManagerError

CERTIFICATE_GENERATION_ERROR

Explanation: This error indicates that the desired certificate could not be created.
Resolution: Inspect the underlying error for details.
Category: SystemInternalAssumptionViolated
Conveyance: This error is logged with log-level ERROR on the server side. This error is exposed on the API with grpc-status INTERNAL without any details due to security reasons
Scaladocs: CERTIFICATE_GENERATION_ERROR

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. This error is exposed on the API with grpc-status ALREADY_EXISTS including a detailed error message
Scaladocs: DUPLICATE_TOPOLOGY_TRANSACTION

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: InvalidIndependentOfSystemState
Conveyance: This error is logged with log-level WARN on the server side. This error is exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message
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. This error is 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. This error is 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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: PUBLIC_KEY_NOT_IN_STORE

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 unusuable.
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. This error is 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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: SECRET_KEY_NOT_IN_STORE

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. This error is exposed on the API with grpc-status INTERNAL without any details due to 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. This error is 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: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: UNAUTHORIZED_TOPOLOGY_TRANSACTION

3.1.1. 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. This error is 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: MaliciousOrFaultyBehaviour
Conveyance: This error is logged with log-level WARN on the server side. This error is exposed on the API with grpc-status UNKNOWN without any details due to security reasons
Scaladocs: FAILED_TO_ADD_MEMBER

PARTICIPANT_NOT_INITIALIZED

Explanation: This error is returned if a domain topology manager attempts to activate a participant without having previously registered the necessary keys.
Resolution: Register the necessary keys and try again.
Category: InvalidGivenCurrentSystemStateOther
Conveyance: This error is logged with log-level INFO on the server side. This error is 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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: WRONG_DOMAIN

3.1.2. 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. This error is 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. This error is 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. This error is 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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: DEPENDENCIES_NOT_VETTED

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. This error is exposed on the API with grpc-status FAILED_PRECONDITION including a detailed error message
Scaladocs: UNINITIALIZED_PARTICIPANT

3.1.3. Domain

3.1.3.1. 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: MaliciousOrFaultyBehaviour
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

3.2. 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. This error is exposed on the API with grpc-status INTERNAL without any details due to security reasons
Scaladocs: TOPOLOGY_DISPATCHING_INTERNAL_ERROR

4. 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

5. 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

6. DatabaseStorageError

DB_STORAGE_DEGRADATION

Explanation: This error indicates that degradation of database storage components.
Resolution: Inspect error message for details.
Category: BackgroundProcessDegradationWarning
Conveyance: This error is logged with log-level WARN on the server side.
Scaladocs: DB_STORAGE_DEGRADATION

7. 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. This error is exposed on the API with grpc-status INVALID_ARGUMENT including a detailed error message
Scaladocs: PROTO_DESERIALIZATION_FAILURE

8. ResilientSequencerSubscription

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

9. 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

Important: This feature is only available in Canton Enterprise

---

3.3.10 High Availability Usage

3.3.10.1 Overview

Canton nodes can be deployed in a highly available manner to ensure that domains and participants will continue operating despite isolated machine failures. See *High Availability* for a detailed description of the architecture in each Canton component to support HA.

3.3.10.2 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 *Domain bootstrapping*.

The domain manager node itself is currently not HA but it is not on the critical path for transaction processing, but for onboarding new parties/participants.

3.3.10.3 HA Setup on Oracle

The HA approach that is used by the participant, mediator, and sequencer nodes requires additional permissions being 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.

3.3.10.4 Mediator

The mediator service uses a hot-standby mechanism, with an arbitrary number of replicas.

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 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 domains initialization.

HA Configuration

HA mediator support is only available in the 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.

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.

3.3.10.5 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*.

```
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
```

(continues on next page)

(continued from previous page)

```
      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. 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 to keep the default value of `10`, as all above explanations are only of theoretical nature 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
  balance roundrobin
  server sequencer1 sequencer1.local:1234 proto h2
  server sequencer2 sequencer2.local:1234 proto h2
  server sequencer3 sequencer3.local:1234 proto h2
```

Client-side load balancing

Using client-side load balancing is recommended where a 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_ha` console command when registering/connecting to the domain:

```
myparticipant.domains.connect_ha(
  "my_domain_alias",
  "https://sequencer1.example.com",
  "https://sequencer2.example.com",
  "https://sequencer3.example.com"
)
```

See the documentation on the `connect` command using a domain connection config for 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.

3.3.10.6 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.

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
        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
```

(continues on next page)

(continued from previous page)

```
    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
    server participant3 participant3.lan:15000 proto h2 check port 8080
```

3.3.11 Identity Management

On-ledger identity management focuses on the distributed aspect of identities between Canton system entities, while user identity management focuses on participants individually 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.

3.3.11.1 Introduction

What is a Canton Identity?

When two system entities such as 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 and to ensure that the recipient can verify the authenticity of the message, or prove the origin of a certain message. 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).

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 is called a *unique identifier (X,N)* 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, and as such, a system entity (or party) is the combination of role (party, participant, mediator, sequencer, domain topology manager) and a unique identifier.

The system entities require knowledge about the keys which will be used for encryption and signing by the respective other entities and this knowledge needs to be shared among all affected system 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.

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.

An 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 an 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*. An topology transaction that refers to a particular unique identifier operates on that namespace and we require that such an 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 to do that. A *namespace delegation* allows to delegate 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* allows to delegate 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.

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.

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` domain service per domain.
6. A domain receives this request and processes it according to the policy (reject, queue, approve). The default setting is approve for convenience during development.
7. The approve policy 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`).

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(
  domainAlias,
  connection,
```

(continues on next page)

(continued from previous page)

```
  manualConnect,
  domainId,
  certificates,
  priority,
  initialRetryDelay,
  maxRetryDelay,
  timeTrackerConfig,
)
```

```
// register the domain configuration
register(config.copy(manualConnect = true))
if (!config.manualConnect) {
  // fetch and confirm domain agreement
  config.sequencerConnection match {
    case _: GrpcSequencerConnection =>
      confirm_agreement(config.domain.unwrap)
    case _ => ()
  }
  reconnect(config.domain.unwrap, retry = false)
  // 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 domain 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 will contact the `DomainService` and check if using the domain service requires the signing of specific terms of services. If required, the terms of service will be displayed to the user and an approval will be locally stored at the participant for later. If approved, the participant will attempt to connect to the domain.
3. The participant opens a GRPC channel to the `DomainService`.
4. The participant verifies that the remote domain is running a protocol version compatible with the participant's version using the `DomainService.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 will download the *domain parameters*, which are the parameters used for the transaction protocol on the particular domain. This happens on every re-connect.
7. The participant sets up an *identity pusher* 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 participant node keys and the registered parties.
8. The domain receives the set of topology transactions of the participant node through the `DomainIdentityService.RegisterIdentityTransaction`. The registration service inspects the validity of the transactions and decides based on the configured domain on-boarding policy. The currently supported policies are `auto-accept`, `queue`, `reject`. While `auto-accept` is convenient for permission-less systems and for development, it will accept any new participant and any topology transaction. The `queue` policy will store the new requested topology transaction in a store and let the domain administrator decide on whether a certain topology transaction will be accepted or not. The policy `reject` will just reject any topology transaction by default. Therefore, whether a participant can join a domain or not is a decision of the domain operator. Canton does not make any assumption on how and when a domain operator decides, but lets the operator implement any process. Note that the currently implemented policies are *experimental* features so far with limited documentation and the final version will allow domain operators to define their own topology transaction request processing policy rather than having to pick from a pre-defined set.
9. The `auto-accept` policy auto-approves all topology transactions as long as they are properly authorised and adds them to the domain topology state. If a new participant appears, the participant is automatically enabled.
10. When a participant is (re-)enabled, the domain identity 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.
11. Now, as 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 *sequencer service*, it can not know what the key is. Therefore, the domain discloses this part of the domain topology state as part of the authorisation challenge.
12. 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.
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.

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 create a namespace key, a signing key and an encryption key for the participant.
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.

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 an 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 managers 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.

3.3.11.2 User Identity Management

Up to here, we covered how on ledger identities are managed. However, every participant needs to manage the access to their local ledger API and be able to permission applications to read or write to that API on behalf of one or more parties. This is done through the appropriate authorization configuration in the ledger API configuration section.

The default implemented authorization will be based on JWT token inspection which can be used with OAuth2, but custom authorization methods can be implemented as a plugin.

Note: This is currently being delivered as an upstream feature and will be exposed as soon as it is available.

3.3.11.3 Cookbook

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.

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:

Domain Initialization

The following steps describe how to manually initialize a domain node:

```
// first, let's create a signing key that is going to control our identity
val identityKey = mydomain.keys.secret.generate_signing_key("default")

// 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("mydomain", namespace)

// create the root certificate for this namespace
mydomain.topology.namespace_delegations.authorize(
  TopologyChangeOp.Add,
  namespace,
  namespace,
  isRootDelegation = true,
)

// set the initial dynamic domain parameters for the domain
mydomain.topology.domain_parameters_changes
  .authorize(DomainId(uid), initialDynamicDomainParameters)

val mediatorId = MediatorId(uid)
Seq[KeyOwner](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.

    // first, create a signing key for this entity
    val signingKey = mydomain.keys.secret.generate_signing_key(
      keyOwner.code.threeLetterId.unwrap + "-signing-key"
    )

    // then, create a topology transaction linking the entity to the signing key
    mydomain.topology.owner_to_key_mappings.authorize(
      TopologyChangeOp.Add,
      keyOwner,
      signingKey.fingerprint,
      KeyPurpose.Signing,
    )
}

// Register the mediator
mydomain.topology.mediator_domain_states.authorize(
  TopologyChangeOp.Add,
  mydomain.id,
  mediatorId,
  RequestSide.Both,
)
```

Participant Initialization

The following steps describe how to manually initialize a participant node:

```
// first, let's create a signing key that is going to control our identity
val identityKey = participant1.keys.secret.generate_signing_key("my-identity")
// use the fingerprint of this key for our identity
val namespace = identityKey.fingerprint

// create the root certificate (self-signed)
participant1.topology.namespace_delegations.authorize(
  TopologyChangeOp.Add,
  namespace,
  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.
participant1.topology.init_id("manualInit", namespace)

// create signing and encryption keys
val enc = participant1.keys.secret.generate_encryption_key()
val sig = participant1.keys.secret.generate_signing_key()

// assign new keys to this participant
Seq(enc, sig).foreach { key =>
  participant1.topology.owner_to_key_mappings.authorize(
    TopologyChangeOp.Add,
    participant1.id,
    key.fingerprint,
    key.purpose,
  )
}
```

Party on two Nodes

Assuming we have party `("Jesper", N1)` which we want to host on two participants: `("participant1", N1)` and `("participant2", N2)`. In this case, we have the party `Jesper` in namespace `N1`, whereas the participant2 is in namespace `N2`. Therefore, we first need to enable the party on the first node, and then we need to authorize the mapping of the party to the participant on both topology managers, as given in below code snippet.

```
// enable party on participant1 (will invoke topology.party_to_participant_
↪mappings.authorize) under the hood
val partyId = participant1.parties.enable("Jesper")
```

(continues on next page)

(continued from previous page)

```
val p2id = participant2.id

// authorize mapping of Jesper to P2 on the topology manager of Jesper
participant1.topology.party_to_participant_mappings.authorize(
  TopologyChangeOp.Add,
  party = partyId, // party unique identifier
  participant = p2id, // participant unique identifier
  side =
    RequestSide.From, // request side is From if signed by the party idm, To if
↪signed by the participant idm.
  permission =
    ParticipantPermission.Submission, // optional argument defaulting to
↪`Submission`.
)
// authorize mapping of Jesper to P2 on the topology manager of P2
participant2.topology.party_to_participant_mappings.authorize(
  TopologyChangeOp.Add,
  partyId,
  p2id,
  side = RequestSide.To,
  permission = ParticipantPermission.Submission,
)
```

Please note however that this currently only works for newly permissioned parties as we don't yet support migrating the current active contract set.

Note that we 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 to create setups where a party is hosted with `Submission` permissions on one node and `Observation` 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 may not learn the identity of the submitting participant. Therefore, a party who delegates `Confirmation` permissions to a participant should trust the participant sufficiently.

Move 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"
```

(continues on next page)

(continued from previous page)

```
val fingerprint = participant1.keys.secret.generate_signing_key(name = name).
↪fingerprint

// create an intermediate certificate authority through a namespace delegation
// 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(namespace, Some(privateKeyFile.toString))

  // delete namespace key (very dangerous ...)
  participant1.keys.secret.delete(namespace, force = true)
```

Later on, when the root namespace key is required for some operations, 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"))
```

Rolling Keys

Canton supports rolling keys during live operation. This works with any key, signing or encryption. In order to ensure continuous operation, we first need to add the new key and then recall the previous key.

```
// Create a new signing key
val newKey = node.keys.secret.generate_signing_key(name)

// Authorize the new key as the new signing key
// The result is that the owner will now have two signing keys, but by convention
↪always the first one added is
// used by everybody.
node.topology.owner_to_key_mappings.authorize(
  TopologyChangeOp.Add,
  owner,
  newKey.fingerprint,
  KeyPurpose.Signing,
)

// wait until new key has appeared everywhere (only relevant if you roll the
↪sequencer key)
eventually() {
  node.keys.public
    .list_by_owner(owner)
```

(continues on next page)

(continued from previous page)

```
    .flatMap(_.signingKeys.map(_.fingerprint)) should contain(newKey.fingerprint)
}

// Remove old key by sending the matching `Remove` transaction
node.topology.owner_to_key_mappings.authorize(
  TopologyChangeOp.Remove,
  owner,
  currentFingerprint,
  KeyPurpose.Signing,
)
```

3.3.12 Monitoring

3.3.12.1 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 which is set to `INFO`, to `DEBUG`. 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. Right now, we only support the `container` profile which logs to STDOUT and turns of flat file logging to avoid storage leaks due to log files within a container.

Please 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`, `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 `--log-level-canton` and `--log-level-root` which can still be used to adjust the log level of the root loggers.

3.3.12.2 Viewing Logs

We strongly recommend the use of a log file viewer such as lnav to view Canton logs and resolve issues. Among other features, lnav has automatic syntax highlighting, convenient filtering for specific log messages, and allows viewing log files of different Canton components in a single view. This makes viewing logs and resolving issues a lot more efficient than simply using standard UNIX tools such as less or grep.

In particular, we have found the following features 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, e.g. with a given trace-id, in, a transaction can be traced across different components, especially when using the single-view-feature described above.
searching for specific log messages (`/<regex>`) and jumping in-between them (`n` and `N`).
automatic syntax highlighting of parts of log messages (e.g. timestamps) and log messages themselves (e.g. `WARN` log messages are yellow).
jumping in-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`).

The custom lnav log format file for Canton logs `canton.lnav.json` is bundled in any Canton release. It can be installed with `lnav -i canton.lnav.json`.

3.3.12.3 Tracing

For further debuggability, Canton provides a trace-id which allows 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 already enabled by default.

It is also possible to configure the service where traces and spans are reported to. Currently Jaeger and Zipkin are supported. For example, Jaeger reporting can be configure as follows:

```
monitoring.tracing.tracer.exporter {
  type = jaeger
  address = ... // default: "localhost"
  port = ... // default: 14250
}
```

It is possible to try it out locally very easily by running Jaeger on a Docker container as follows:

```
docker run --rm -it --name jaeger\
  -p 16686:16686 \
  -p 14250:14250 \
  jaegertracing/all-in-one:1.22.0
```

Sampling

It is also possible to change how often spans are sampled (i.e. reported to the configured exporter). By default it will always report (`monitoring.tracing.tracer.sampler.type = always-on`). It can also be configured to never report (`monitoring.tracing.tracer.sampler.type = always-off`, although not super useful). And it can also be configured so that a specific fraction of spans are reported like below:

```
monitoring.tracing.tracer.sampler = {
  type = trace-id-ratio
  ratio = 0.5
}
```

There is one last property of sampling that can be optionally changed. By default we have parent-based sampling on (`monitoring.tracing.tracer.sampler.parent-based = true`) which means that a span is sampled iff its parent is sampled (the root span will follow the configured sampling strategy). This way, there will never be incomplete traces, so either the full trace is sampled or not. If this property is changed, all spans will follow the configured sampling strategy ignoring whether the parent is sampled or not.

Known Limitations

Not every trace created which can currently be observed in logs are reported to the configured trace collector service. Traces originated at console commands or that are part of the transaction protocol are largely well reported, while other kinds of traces are being added to the set of reported traces as the need arise.

Also, even the transaction protocol trace has a know limitation which is that once some command is submitted (and its trace fully reported), if there are any resulting daml events which are subsequently processed as a result, a new trace is created as currently the ledger api does not propagate any trace context info from command submission to transaction subscription. This can be observed for example by the fact that if a participant creates a ping command, it is possible to see the full transaction processing trace of the ping command being submitted, but then the participant which processes the ping by creating a pong command will then create a separate trace instead of continuing to use the same one.

3.3.12.4 Status

Each Canton node exposes rich status information. Running

```
<node>.health.status
```

will return a status object which can be one of

`Failure` - if the status of the node can not be determined, including an error message why it failed
`NotInitialized` - if the node is not yet initialized
`Success[NodeStatus]` - if the status could be determined including the detailed status.

Depending on the node type, the `NodeStatus` will differ. A participant node will respond 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:` - list of domains the participant is currently connected to properly
`Unhealthy domains:` - list of domains the participant is trying to be connected to but where the connection is not ready for command submission.
`Active:` - true if this instance is the active replica (can be false in case of the passive instance of a high-availability deployment)

A domain node or a sequencer node will respond 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 if the embedded sequencer writer is operational

A domain topology manager or a mediator node will return

`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 (can be false in case of the passive instance of a high-availability deployment)

3.3.12.5 Health Dumps

In order to provide efficient support, we need as much information as possible. For this purpose, Canton implements an information gathering facility that will gather key essential system information for our support staff. Therefore, if you encounter an error where you need our help, please ensure the following:

Start Canton in interactive mode, with the `-v` option to enable debug logging: `./bin/canton -v -c <myconfig>`. This will provide you with 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 the ZIP file.

This will create 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 logfile. We don't log overly sensitive data into log files.
A current snapshot on Canton metrics.
A stacktrace for each running thread.

Please provide the gathered information together with the exact list of steps you did that lead to the issue to your support contact. Providing complete information is very important to us in order to help you troubleshoot your issues.

3.3.12.6 Health Check

The `canton` process can optionally expose an HTTP endpoint indicating if the process believes it is healthy. This is intended for use in uptime checks and liveness probes. If enabled, the `/health` endpoint will respond to a `GET` http request with a 200 HTTP status code if healthy or 500 if unhealthy (with a plain text description of why it is unhealthy).

To enable this health endpoint add a `monitoring` section to the canton configuration. As this health check is for the whole process, it is added 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 will have `participant1` ledger ping itself every 30 seconds. The process will be considered healthy if the ping is successful.

3.3.12.7 Metrics

Canton uses dropwizard's metrics library to report metrics. The metrics library supports a variety of reporting backends. JMX based reporting (only for testing purposes) 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
```

(continues on next page)

(continued from previous page)

```
  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"
  }]
}]
```

or reported via Prometheus (to Grafana) using

```
canton.monitoring.metrics.reporters = [{
  type = prometheus
  address = "localhost" // default
  port = 9000 // default
}]
```

When using the `graphite` or `csv` reporters, Canton will periodically evaluate all metrics matching the given filters. It is therefore advisable to 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
```

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: Gauge

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

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

canton.<domain>.protocol-messages.confirmation-request-size

Summary: Confirmation request size
Description: Records the histogram of the sizes of (transaction) confirmation requests.
Type: Histogram

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

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: Gauge

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

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

canton.<domain>.sequencer-client.delay

Summary: The delay on the event processing
Description: Every message received from the sequencer carries a timestamp. The delay provides the difference between the sequencing time and the processing time. The difference can be a result of either clock-skew or if the system is overloaded and doesn't manage to keep up with processing events.
Type: Gauge

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

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

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

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: Gauge

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

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

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

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

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

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

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

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

canton.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: Gauge

canton.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: Gauge

canton.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

canton.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: Gauge

canton.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: Gauge

canton.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

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

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

Domain Metrics

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

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

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

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

canton.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: Gauge

canton.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: Gauge

canton.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

canton.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: Gauge

canton.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: Gauge

canton.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

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

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

canton.mediator.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

canton.mediator.sequencer-client.delay

Summary: The delay on the event processing
Description: Every message received from the sequencer carries a timestamp. The delay provides the difference between the sequencing time and the processing time. The difference can be a result of either clock-skew or if the system is overloaded and doesn't manage to keep up with processing events.
Type: Gauge

canton.mediator.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

canton.mediator.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

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

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: Gauge

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

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

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

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

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

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

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

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: Gauge

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: Gauge

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

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: Gauge

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: Gauge

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

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

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.
Type: Meter

canton.sequencer.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

canton.sequencer.sequencer-client.delay

Summary: The delay on the event processing
Description: Every message received from the sequencer carries a timestamp. The delay provides the difference between the sequencing time and the processing time. The difference can be a result of either clock-skew or if the system is overloaded and doesn't manage to keep up with processing events.
Type: Gauge

canton.sequencer.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

canton.sequencer.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

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

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: Gauge

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

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

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

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

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

canton.topology-manager.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

canton.topology-manager.sequencer-client.delay

Summary: The delay on the event processing
Description: Every message received from the sequencer carries a timestamp. The delay provides the difference between the sequencing time and the processing time. The difference can be a result of either clock-skew or if the system is overloaded and doesn't manage to keep up with processing events.
Type: Gauge

canton.topology-manager.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

canton.topology-manager.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

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

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: Gauge

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

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

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

3.3.13 Operational Processes

3.3.13.1 Managing domain entities

Domain bootstrapping

If you're running a domain node in its default configuration, it will have a sequence and mediator embedded and these components will be automatically bootstrapped for you.

However if your domain operates with external sequencers and mediators for improved availability and performance properties, you need to instead 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 as illustrated below:

```
domainManager1.setup.bootstrap_domain(Seq(sequencer1), Seq(mediator1))
```

Domain managers are configured as `domain-managers` under the `canton` configuration. Domain managers are configured similarly to domain nodes, except that there is 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 other non-database based sequencer such as Ethereum or Fabric sequencers you need to have each node initialized individually. For these kinds of sequencers you can either initialize them as part of the initial domain bootstrap shown above or you can dynamically add a new sequencer at a later point like follows:

```
domainManager1.setup.onboard_new_sequencer(
  initialSequencer = sequencer1,
  newSequencer = sequencer2,
)
```

3.3.13.2 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**.

A participant can get the current parameters on a domain it is connected to using the following command:

```
participant.topology.domain_parameters_changes.get_latest(mydomain.id)
```

A domain operator can update some of the parameters as follows:

```
mydomain.service.update_dynamic_parameters(_.copy(
  participantResponseTimeout = TimeoutDuration.ofSeconds(10)
))
```

3.3.13.3 Importing existing Contracts

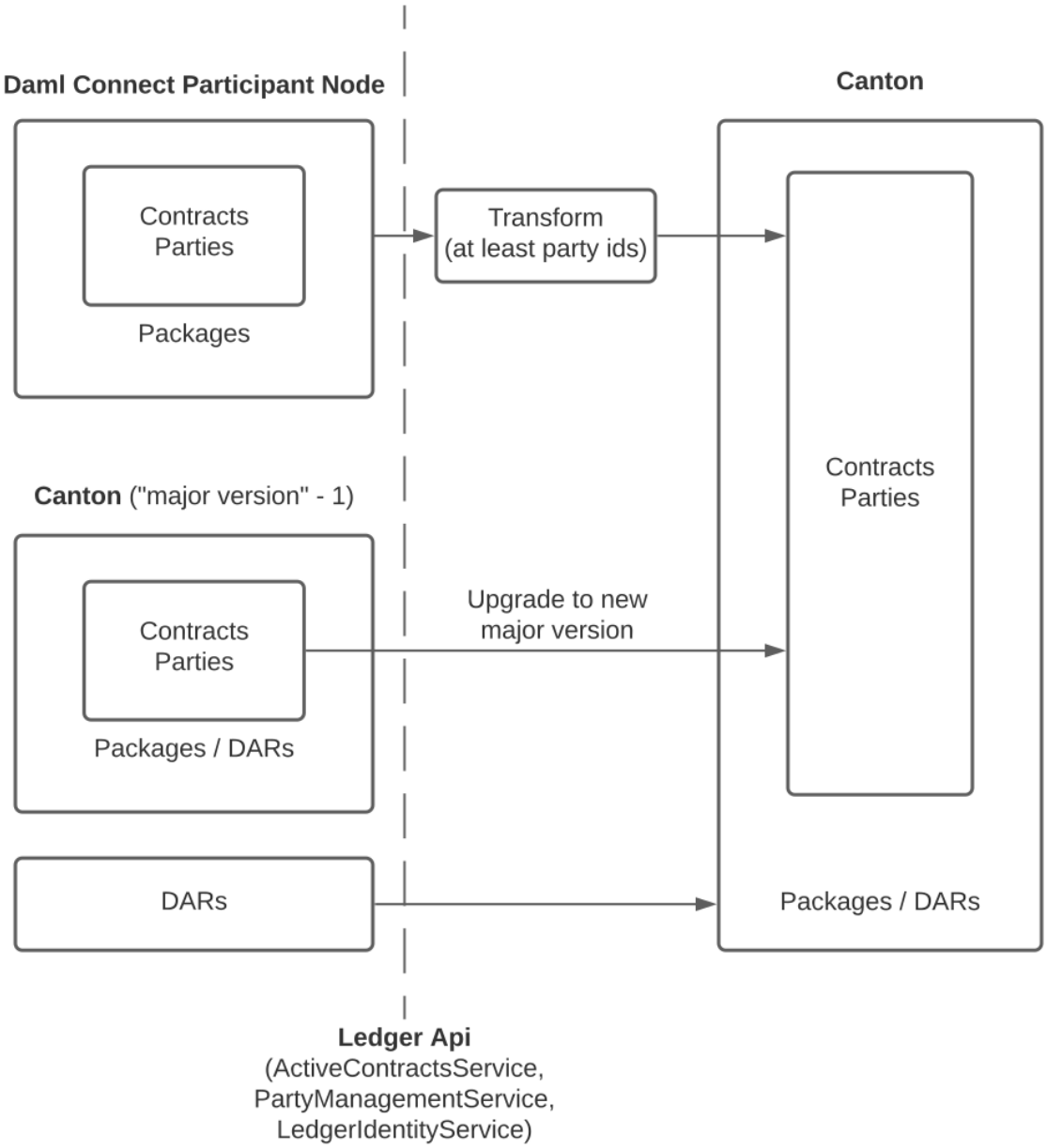
You may have existing contracts, parties, and DARs in other Daml Participant Nodes (such as the Daml sandbox) that you want to import into your Canton-based participant node. To address this need, you can extract contracts and associated parties via the ledger api, modify contracts, parties, and daml 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.

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.


Daml Connect Participant Node
Contracts
Parties
Packages
Transform
(at least party ids)
Canton
Contracts
Parties
Canton ("major version" - 1)
Contracts
Parties
Packages / DARs
Upgrade to new
major version
DARs
Packages / DARs
Ledger Api
(ActiveContractsService,
PartyManagementService,
LedgerIdentityService)

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).

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 \
  -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())
    localParties = allParties.collect {
      case PartyDetails(party, _, isLocal) if isLocal => LfPartyId.
↪assertFromString(party)
    }
```

(continues on next page)

(continued from previous page)

```
    // Look up the ledger id needed next to query for the contracts.
    ledgerId <- queryLedgerApi(LedgerApiCommands.LedgerIdentityService.
↪GetLedgerIdentity())

    // Query the ActiveContractsService for the actual contracts
    acs <- queryLedgerApi(
      LedgerApiCommands.AcsService.GetActiveContracts(ledgerId, localParties.
↪toSet)
    )
  } 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

  acs.map { event =>
    ValueRemapper.convertEvent(identity, stripPartyIdSuffix)(event)
  }
}

def lookUpPartyId(participant: ParticipantReference, party: String): PartyId =
  participant.parties.list(filterParty = party + SafeSimpleString.delimiter).map(_
↪.party).head
```

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).

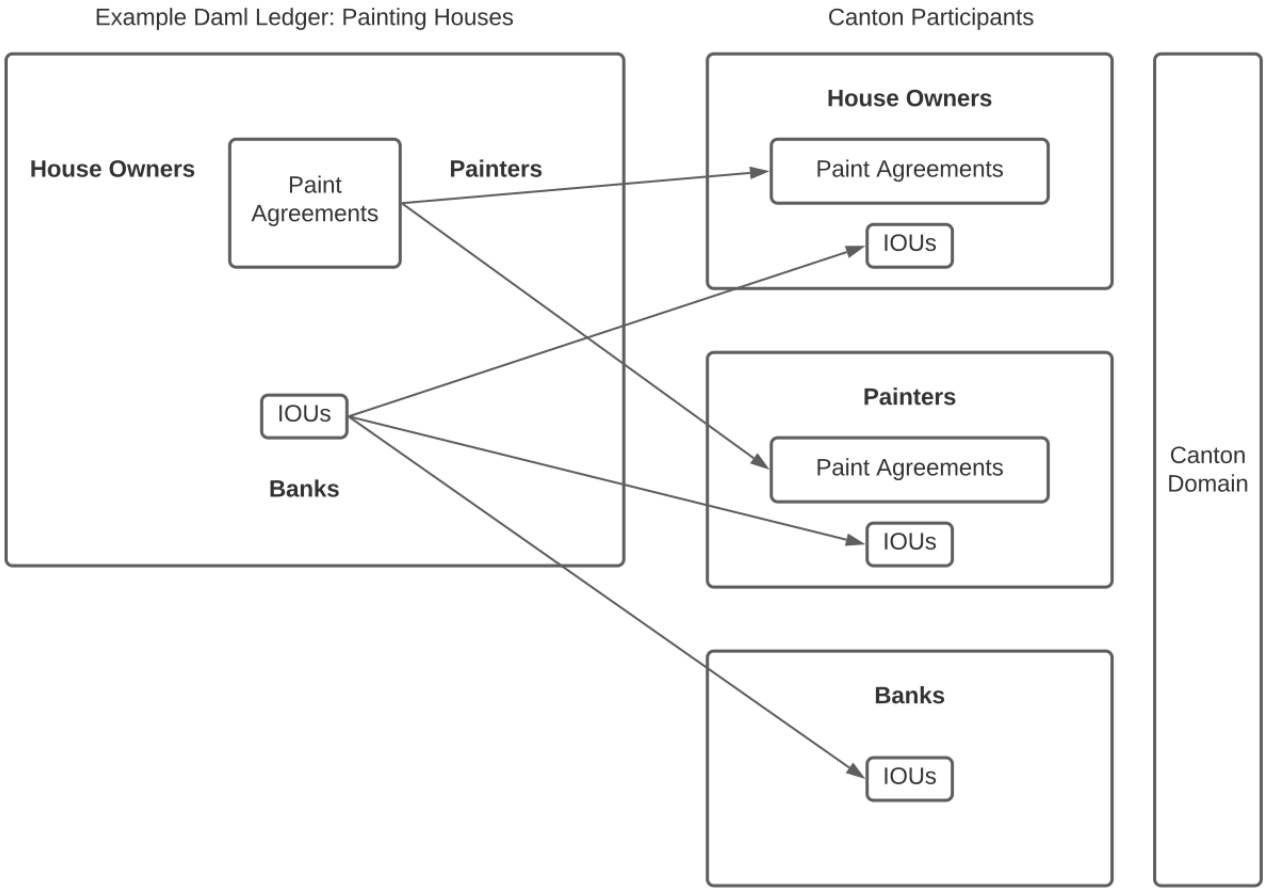
```
val acs =
  queryActiveContractsFromDamlLedger(
    exportLedger.config.ledgerApi.address,
    exportLedger.config.ledgerApi.port,
    exportLedger.config.ledgerApi.tls.map(_.clientConfig),
  )

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)
```

(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)

```
// 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

// 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)
    .get
    .clock
    .now
    .toInstant

// Filter active contracts based on participant parties and upload.
partyAssignments.foreach { case (participant, rawParties) =>
  val parties = rawParties.map(toCantonParty(_))
  val participantAcs = acsToImportToCanton
    .collect {
      case event
          if event.signatories.intersect(parties).nonEmpty
            || event.observers.intersect(parties).nonEmpty
            || event.witnessParties.intersect(parties).nonEmpty =>
        val wrappedCreatedEvent = WrappedCreatedEvent(event)

        SerializableContractWithWitnesses(
```

(continues on next page)

(continued from previous page)

```
          utils
            .contract_data_to_instance(wrappedCreatedEvent.toContractData,
↪ledgerCreateTime),
          Set.empty,
        )
    }

  participant.repair.add(importLedgerDomain.name, participantAcs,
↪ignoreAlreadyAdded = false)
}

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,
```

(continues on next page)

(continued from previous page)

```
): 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
    .exerciseAcceptByOwner(houseOwner.toPrim, 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)
  paintHouseContracts
}
```

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.

3.3.13.4 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 considering the domain's backup is more recent than that of the participant's. 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 backup both participant and domain, always backup participant database before the domain.

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 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*

We recommend that in production, a domain should be run with offsite synchronous replication to assure the most crucial data is always safely backed up and as up-to-date as possible.

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

3.3.13.5 Database Failover

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. If the data in one replica gets lost, the database can still failover to another replica **without any data loss.** 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.

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.

3.3.13.6 Ledger Pruning

Pruning the ledger frees up storage space by deleting state no longer needed by participants, domain sequencers, and mediators. It also serves as a mechanism to help implement right-to-forget mandates such as GDPR.

The following commands allow you to prune events and inactive contracts up to a specified time from the various components:

Prune participants via the *prune* command specifying a ledger offset obtained by specifying a timestamp received by a call to get_offset_by_time .
Prune domain sequencers and mediators via their respective *prune_at* commands.

The pruning operations impact the regular workload (lowering throughput during pruning by as much as 50% in our test environments), so depending on your requirements it might make sense to schedule pruning at off-peak times or during maintenance windows such as after taking database backups.

The following canton console code illustrates best practices such as :

Error handling ensures that pruning errors raise an alert. Catching the *CommandFailure* exception also ensures that a problem encountered while pruning one component still lets pruning other components proceed allowing corresponding storage to be freed up.
Pruning one node at a time rather than all nodes in parallel somewhat limits the impact on concurrently executing workload. If you configure pruning to run during a maintenance window with no concurrent workload, and as long as the database backend has sufficient capacity, you may prune participants and domains in parallel.

```
import com.digitalasset.canton.console.{CommandFailure, ParticipantReference}
import com.digitalasset.canton.data.CantonTimestamp
import java.time.Duration

def pruneAllNodes(pruneUpToIncluding: CantonTimestamp)(implicit env:
↪ConsoleEnvironment): Unit = {
  import env._

  // If pruning a particular component fails, alert the user, but proceed pruning
↪other components.
  // Therefore prune failures in one component still allow other components to be
↪pruned
  // minimizing the chance of running out of overall storage space.
  def alertOnErrorButMoveOn(
      component: String,
      ts: CantonTimestamp,
      invokePruning: CantonTimestamp => Unit,
  ): Unit =
    try {
      invokePruning(ts)
    } catch {
      case _: CommandFailure =>
        logger.warn(
          s"Error pruning ${component} up to ${ts}. See previous log error for
↪details. Moving on..."
        )
    }
```

(continues on next page)

(continued from previous page)

```
  // Helper to prune a participant by time for consistency with domain prune
↪signatures
  def pruneParticipantAt(p: ParticipantReference)(pruneUpToIncluding:
↪CantonTimestamp): Unit = {
    val pruneUpToOffset = p.pruning.get_offset_by_time(pruneUpToIncluding.
↪toInstant)
    pruneUpToOffset match {
      case Some(offset) => p.pruning.prune(offset)
      case None => logger.info(s"Nothing to prune up to ${pruneUpToIncluding}")
    }
  }

  val participantsToPrune = participants.all
  val domainsToPrune = domains.all

  // Prune all nodes one after the other rather than in parallel to limit the
↪impact on concurrent workload.
  participantsToPrune.foreach(participant =>
    alertOnErrorButMoveOn(participant.name, pruneUpToIncluding,
↪pruneParticipantAt(participant))
  )

  domainsToPrune.foreach { domain =>
    alertOnErrorButMoveOn(
      s"${domain.name} sequencer",
      pruneUpToIncluding,
      domain.sequencer.pruning.prune_at,
    )
    alertOnErrorButMoveOn(
      s"${domain.name} mediator",
      pruneUpToIncluding,
      domain.mediator.prune_at,
    )
  }
}
```

Invoke pruning from within your scheduling environment and by specifying the ledger data retention period like so:

```
val retainMostRecent = Duration.ofDays(30)
pruneAllNodes(CantonTimestamp.now().minus(retainMostRecent))
```

Pruning Ledgers in Test Environments

While it is a best practice for test environments to match production configurations, testing pruning involves challenges related to the amount of retained data:

> Test environments may not have the same amount of storage space to hold data volumes present in production.
> It may be impractical to wait long enough until test environments have accrued data to expected production retention times that are often measured in months.

As a result you may choose to prune test environments more aggressively. When using databases other than Oracle with a lower retention time, use the same code as when pruning production. On

Oracle however you may observe performance degradation when pruning the majority of the ledger data in one go. In such cases breaking up pruning invocations into multiple chunks likely speeds up pruning:

```
// An example test environment configuration in which hardly any data is retained.
val pruningFrequency = Duration.ofDays(1)
val retainMostRecent = Duration.ofMinutes(20)
val pruningStartedAt = CantonTimestamp.now()
val isOracle = true

// Deleting the majority of rows from an Oracle table has been observed to
// take a long time. Avoid non-linear performance degradation by breaking up one
↪prune call into
// several calls with progressively more recent pruning timestamps.
if (isOracle && retainMostRecent.compareTo(pruningFrequency) < 0) {
  val numChunks = 8L
  val delta = pruningFrequency.minus(retainMostRecent).dividedBy(numChunks)
  for (chunk <- 1L to numChunks) yield {
    val chunkRetentionTimestamp = pruningFrequency.minus(delta.
↪multipliedBy(chunk))
    pruneAllNodes(pruningStartedAt.minus(chunkRetentionTimestamp))
  }
}

pruneAllNodes(pruningStartedAt.minus(retainMostRecent))
```

3.3.13.7 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.

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.

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)
  )
}

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(
  _.filter {
    case (_isActive, contract)
        if contract.metadata.stakeholders
          .contains(Alice.toLf) && contract.metadata.stakeholders.contains(Bob.
↪toLf) =>
      true
    case _ => false
  }.toSet
)

clue("checking if contracts match") {
  sharedContracts1 shouldBe sharedContracts2
```

(continues on next page)

(continued from previous page)

```
}

// 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.

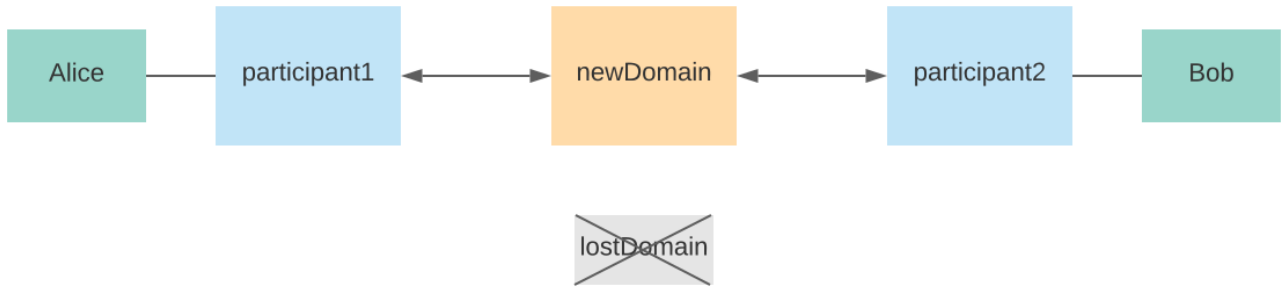
Once each participant has associated the contracts with newDomain, let's have them reconnect, and we should be able to confirm that the new domain is able to execute workflows from where the lost domain disappeared.

```
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)) =>
    participant.ledger_api.commands.submit_flat(
      Seq(payer),
      Seq(
        Iou
          .Iou(
            payer.toPrim,
            owner.toPrim,
            Iou.Amount(value = 200, currency = "USD"),
```

(continues on next page)

(continued from previous page)

```
            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(owner.toPrim).
↪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.

3.3.14 Versioning

3.3.14.1 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)
- Non-preview and non-testing Admin API & Console commands
- 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*).

3.3.14.2 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.

3.3.14.3 For Canton participant and domain operators

In addition to the Canton release version, the Canton protocol version is another important version for participant and domain operators. It has major, minor and patch digits.

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 highest protocol version it supports and supports all previous protocol versions of the same major version. 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 1.3.0 will be able to connect to all domains configured to use protocol version <= 1.3.0. It won't be able to connect to a domain configured to use protocol version > 1.3.0. As a result, minor and patch version upgrades of Canton components can be done independently without any loss of interoperability.

To see the highest protocol version a Canton component supports (e.g., 1.5.0), run

```
canton --version
```

(where `canton` is an alias for the path pointing to the Canton release binary `bin/canton`).

Configuring the protocol version

A Canton driver or domain operator is able to configure the protocol version spoken on the domain (e.g. 1.3.2). 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 1.3.0, participants that only support protocol version 1.2.0 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 1.2.0 or lower. Note that this is always possible in such a scenario because if a domains supports protocol version 1.3.0, it also supports protocol version 1.2.0.

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.

3.3.15 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.

3.3.15.1 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.

3.3.15.2 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.

3.3.15.3 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 it's 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 can not 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(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(cryptoPureApi(participant1.
↪config))(_)
        }


- Some classes do not have a (public) ``toProto*`` method, but they can be
↪serialized to a
  `ByteString <https://developers.google.com/protocol-buffers/docs/reference/java/
↪com/google/protobuf/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,
```

(continues on next page)

(continued from previous page)

```
    CantonTimestamp.MaxValue.toInstant,
  )
  .lastOption
  .getOrElse(
    throw new NoSuchElementException("Unable to find an acs commitment.")
  )
  .message

// Dump the commitment to a file.
utils.write_to_file(
  lastCommitment.toByteString(ProtocolVersion.default),
  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)
```

3.3.15.4 How to Setup Canton to Get Best Performance?

In this section, the findings from our internal performance tests are outlined to help you achieve best performance for your Canton application.

System Design / Architecture

Make sure to use Canton Enterprise because it 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 would become a bottleneck of 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 / exercise) 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*.

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). Prefer hosting a Canton node and its database on the same machine. This is likely faster than running several Canton nodes on the same machine and the databases on a separate machine; for, the latency between Canton nodes is much less performance critical 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.

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_parameters(
  _.copy(
    participantResponseTimeout = TimeoutDuration.ofSeconds(60),
    mediatorReactionTimeout = TimeoutDuration.ofSeconds(60),
  )
)
```

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.

Configure generous resource limits. Resource limits are used to prevent ledger applications from overloading Canton by sending commands at an excessive rate. While they may be required to protect the system from denial of service attacks in a production environment, they can get in the way when doing performance measurements. Resource limits can be configured as follows from the Canton console:

```
participant1.resources.set_resource_limits(
  ResourceLimits(
    maxDirtyRequests = Some(10000),
    maxRate = Some(10000),
  )
)
```

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. Detailed instructions can be found in the Section *Max Connection Settings*.

Size of database task queue. If you are seeing frequent `RejectedExecutionException`s when Canton queries the database, increase the size of the task queue, as described in Section *Database task queue full*.

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.synchronous-commit-mode = off
```

Log level. Make sure that Canton outputs log messages only at level INFO and above.

Disable additional consistency checks. Additional consistency-checks would degrade performance substantially. Make sure they are disabled by including the following line into your Canton configuration:

```
canton.parameters.enable-additional-consistency-checks = false
```

3.3.15.5 Why is Canton complaining about my database version?

Postgres

Canton is tested with Postgres 11, so this is the recommended version. Canton is also likely to work with higher versions, but will WARN when a higher version is encountered. By default, Canton will not start when the Postgres version is below 11.

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:

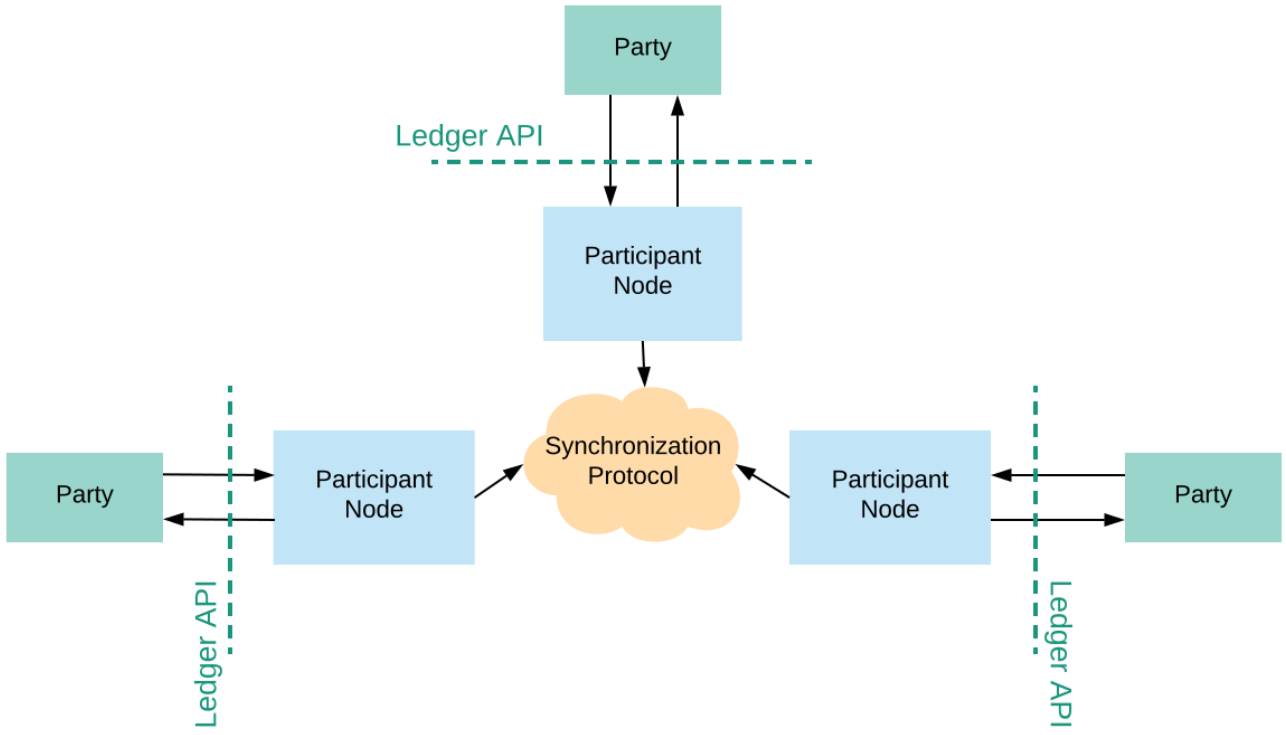
```
canton.parameters.non-standard-config = "yes"
```

Note that this will disable all standard config checks, not just those for the database.

3.4 Architecture In-Depth

3.4.1 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.

3.4.1.1 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. This can be improved in future versions by employing multiple participant nodes.
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 system is overloaded or in case of network failures, the bound can be exceeded. This can be improved in future versions by employing multiple participant nodes.
Design limitation: The transparency requirement can be violated if the submitter node is Byzantine. In particular, it can happen that I learn about the existence of these actions, but not about their contents (including the contracts used).
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.
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.

Automatic Daml package distribution. I want the system to notify my counterparties about my uploaded Daml packages 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.

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.

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 nods. 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*)

3.4.1.2 Resource limits

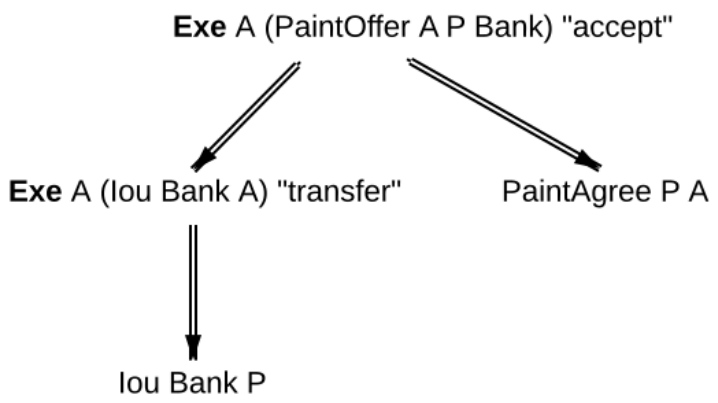
This section specifies upper bounds on the sizes of data structures. The system must be able to process data structures within the given size limits.

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 developers 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.

3.4.1.3 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.

Compliance. I want the system to be compliant with international regulations.

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 my 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.

Garbage collection. I want the system to provide garbage collection 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. node's the 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.
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.
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.
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.
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.

Note: This item is not yet implemented.

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.
Backwards and forward protocol compatibility within a major version. I want system components supporting the same major version of the protocol to be able to communicate seamlessly.
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.
Cross-version backward and forward 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. Note that the requirement does not apply to completely different synchronization protocols (e.g., Daml on SQL and Canton).

Note: This item is not yet implemented.

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.

Note: This item is not yet implemented.

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.
This item is scheduled on the Daml roadmap.

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.
This item is scheduled on the roadmap.

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.
This item is scheduled on the roadmap.

Participant compromise recovery. I want to have a procedure in place that can be followed to recover from a malfunctioning or a compromised 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.

Note: This item is not yet implemented.

Domain entity compromise recovery. I want to have a procedure in place that can be followed to recover a compromised domain entity, so that the system guarantees can be restored after the procedure is complete.

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.

Note: This item is not yet implemented.

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.

Note: This item is not yet implemented.

Adding parties to participants. I want to be able to start using the DA system at any point in time, by choosing to use a new or an already existing participant node.
Identity provider integration. I want the synchronization protocol to integrate with an identity provider service, so that I can use this service to manage the party-to-participant and participant-to-cryptographic-keys mappings.
Identity information updates. I want the synchronization protocol to track updates by the identity provider service, 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 need not provide me with the ledger changes prior to migration.

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.
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 communication. I want the system to provide metrics on the state of all communication links 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 is thoroughly tested 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 is thoroughly tested to be able to detect and recover from faulty behaviour of domain components, such that occasional issues don't break the system permanently.

Note: This item is not yet implemented.

3.4.1.4 Known limitations

In this section, we explain current limitations of Canton that we intend to overcome in future versions. Requirements that have been marked as not implemented or scheduled on the roadmap are not repeated in this section.

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.
Cryptographic evidence extraction: There is currently no public tooling to extract cryptographic evidence for audit and legal actions.

Reliability

Data store consistency: There is no tooling for verifying the consistency of data stores. This tooling would allow users to double-check if crash recovery has recovered a node to a consistent state.
Exceeding resource limits: We have not yet tested systematically whether Canton always fails gracefully, if its resource limits are exceeded.
H2 support: The H2 database backend is not supported for production scenarios.

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.
Data store content upgradeability: We version and manage data stores, but as the product is unfinished, we take the freedom to optimize the stores without providing data continuity.
Protocol upgradeability: We check the protocol version and refuse to operate if the protocol versions mismatch. We do not yet support running nodes with multiple major versions of the protocol so that nodes can be upgraded one by one. To upgrade the Canton version of a node, the node currently needs to be shutdown.

Security

No resilience to dishonest submitters: We have not yet implemented all planned validations on incoming requests. Therefore, compromise of a submitter participant may remain undetected and get the system into an inconsistent state. Consequently, if Canton is run across organizations, these organizations need to mutually trust each other. As part of our future roadmap, we will implement the missing validations.
Denial of service attacks: We have not yet systematically implemented countermeasures to denial of service attacks. E.g., a faulty or malicious participant may overload the system by sending large numbers of messages.
Information leakage: As the product is unfinished, we sometimes take the freedom to write contract data to log files. As part of our future roadmap, we will systematically ensure that we do not leak confidential information in unexpected ways.
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

Reliability

Crash recovery: Both the domain and participants can generally recover from a crash to a consistent state. Currently, we cannot exclude the possibility that recovery may fail, as we might have not yet tested every possible scenario. As part of our future roadmap, we will perform the required testing and hardening. In the meantime, the set of repair commands allows an administrator to manually address any issues resulting from failed recovery.
Sub-component health monitoring: Components are not yet systematically monitored and may therefore not shutdown in case of faulty behavior.

System hardening: Canton is designed to deal gracefully with hazardous events such as network or database outages. However, there may be scenarios that we have not yet covered.
Unbounded decision time: A participant strives for delivering a command completion for every command that has been submitted. In every distributed system it may occur that the system does not produce a response to a request (e.g. due to network or database outages). Consequently, it may happen that Canton does not output a command completion for a submitted command.
Unnecessary rejections: Under adverse conditions (e.g. database or network outages, high load), Canton may reject commands that it would otherwise accept.
Clean shutdown: We cannot yet exclude the possibility that a node reports errors if it is shutdown while processing requests. As part of our roadmap, we will perform further hardening and testing.

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 is an experimental feature. As part of our future roadmap, we plan to perform further hardening and testing, including improvements to user experience and performance. The public API does not yet allow for pruning transfers, transferred contracts, parties, participants, domains, DARs or packages that are no longer in use.
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.
Error messages: On invalid user input or configuration, Canton will output an error message. Sometimes the error message is not yet as descriptive as it could be.
The **Canton documentation** is quite extensive but may require some restructuring for readability.
Minor version compatibility: We do not yet exhaustively test whether different minor Canton versions are compatible.

Performance

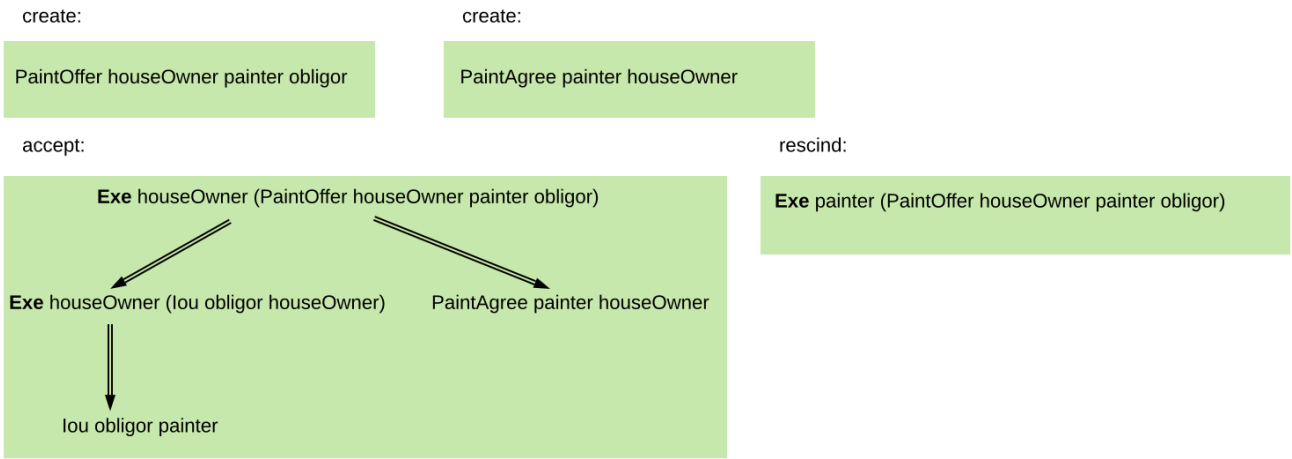
Performance tuning: While we have made sure that the architecture can deliver high throughput numbers, we are still in the process of improving the efficency of our implementation, increasing the throughput.

3.4.1.5 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

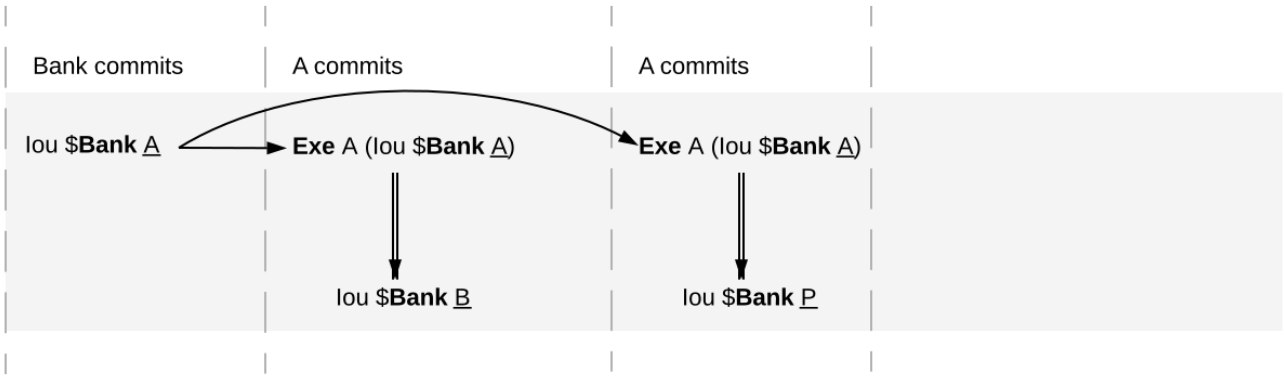
The validity requirement on the ledger made an exception for the *consistency* of contracts without honestly represented signatories. We explain the exception using the paint offer example from the *ledger model*. Recall that the example assumed contracts of the form *PaintOffer houseOwner painter obligor* 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.

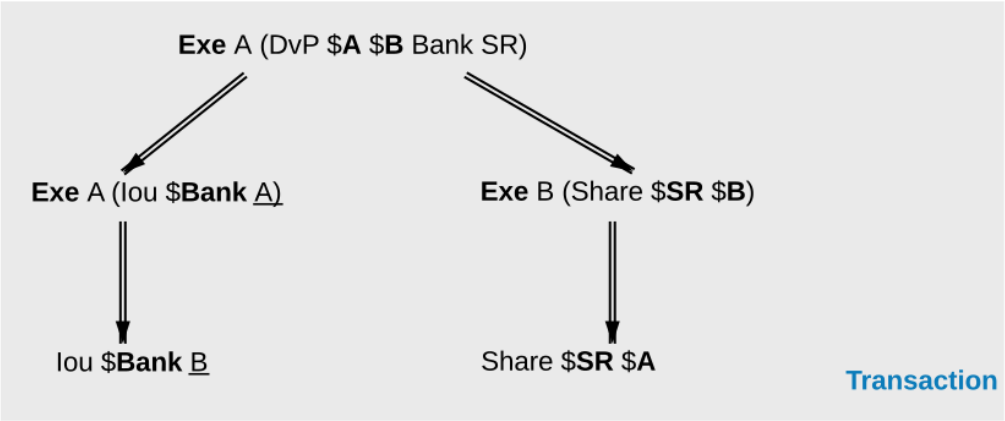
Crashes and Byzantine behavior can inhibit liveness. To cope, the so-called VIP confirmation policy allows any trusted participant to add change requests to the ledger without the involvement of other parties. This policy can be used in settings where there is a central trusted party. Today's financial markets are an example of such a setting.

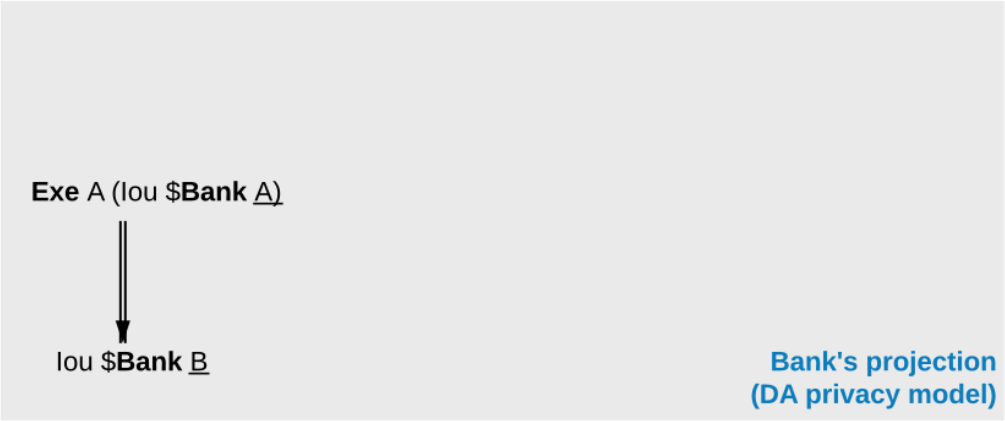
The no-rejection guarantees can be further improved by constructing Daml models that ensure that the submitter is a stakeholder on all contracts in a transaction. That way, rejects due to Byzantine behavior of other participants can be detected by the submitter. Furthermore, if necessary, the synchronization service itself could be changed to improve its properties in a future version, by including so-called bounded timeout extensions and attestators.

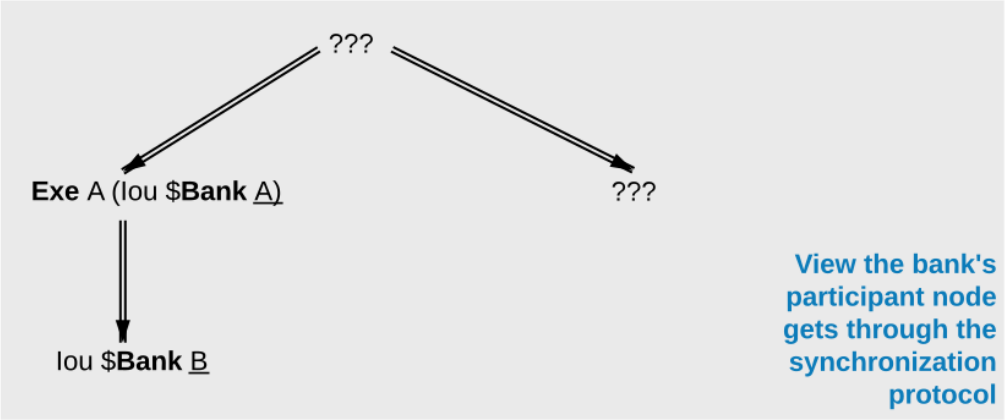
Privacy

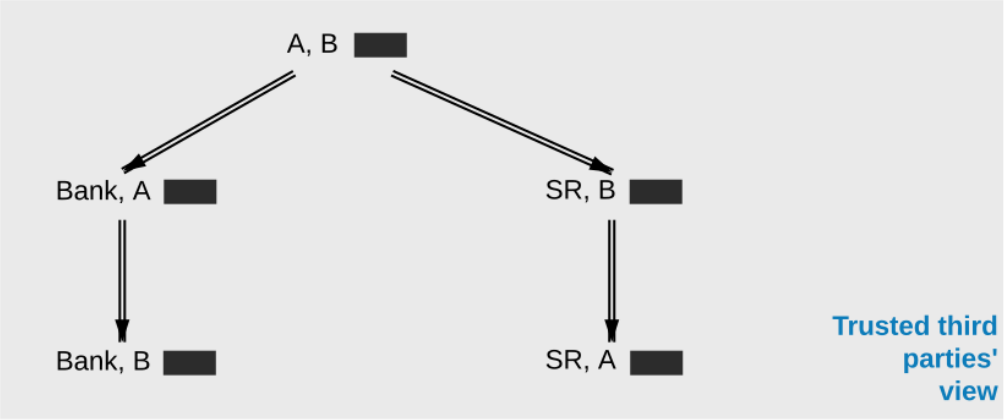
Consider a transaction where Alice buys some shares from Bob (a delivery-versus-payment transaction). The shares are registered at the share registry SR, and Alice is paying with an IOU issued to her by a bank. We depict the transaction in the first image below. Next, we show the bank's projection of 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

3.4.2 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.

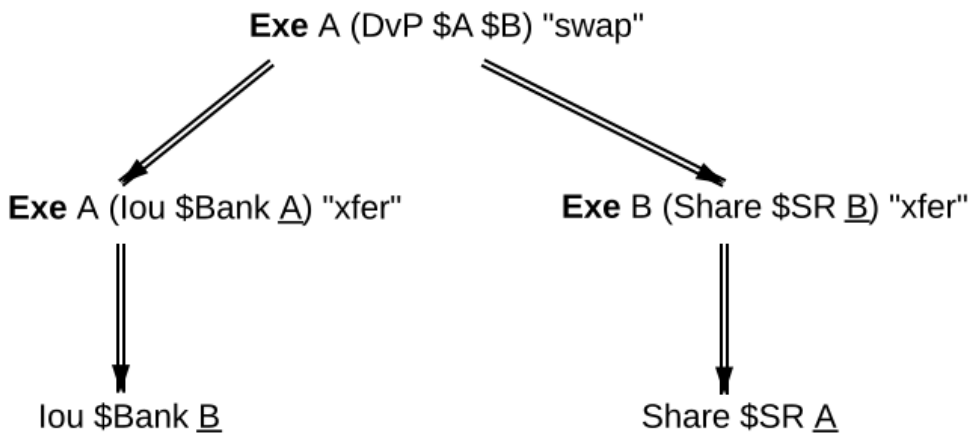
3.4.2.1 Canton 101

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.

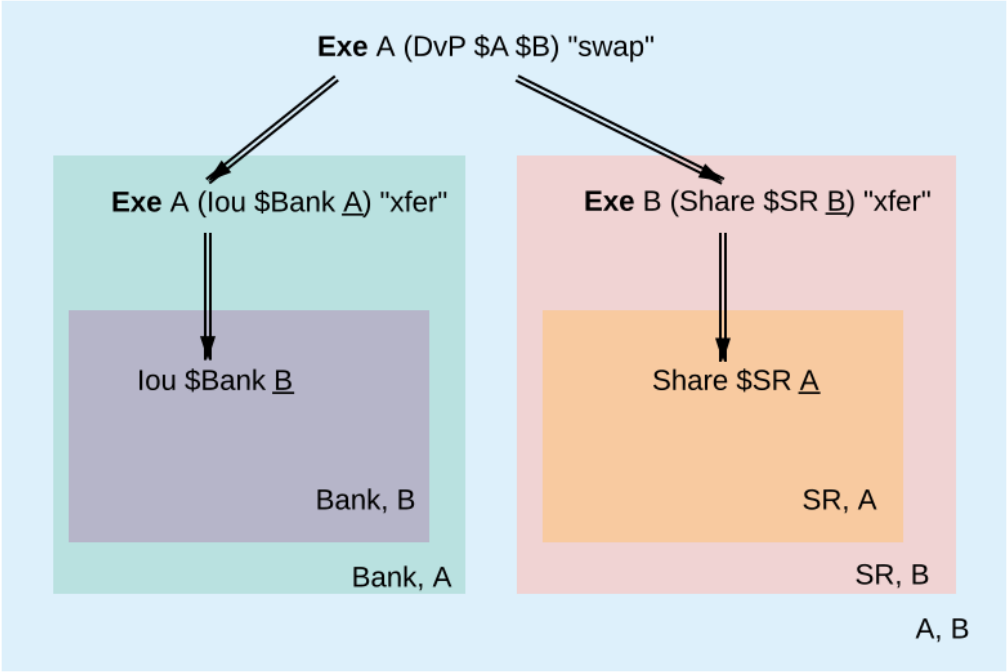


Fig. 2: Views in the transaction; each box represents a transaction part visible to the participants in its bottom-right corner. A participant might receive several views, some of which can be nested.

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 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.

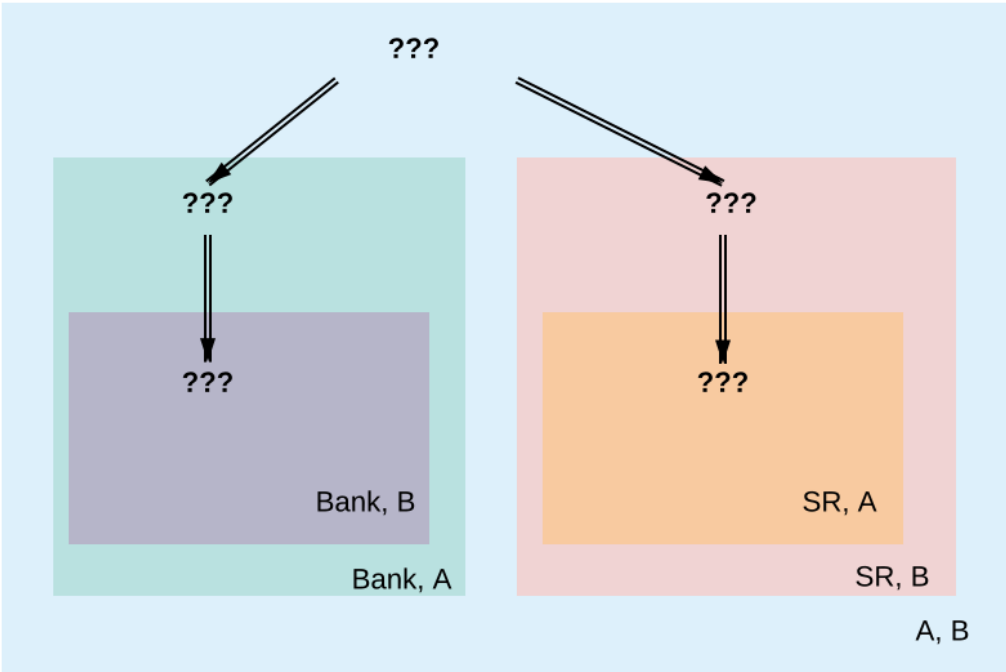


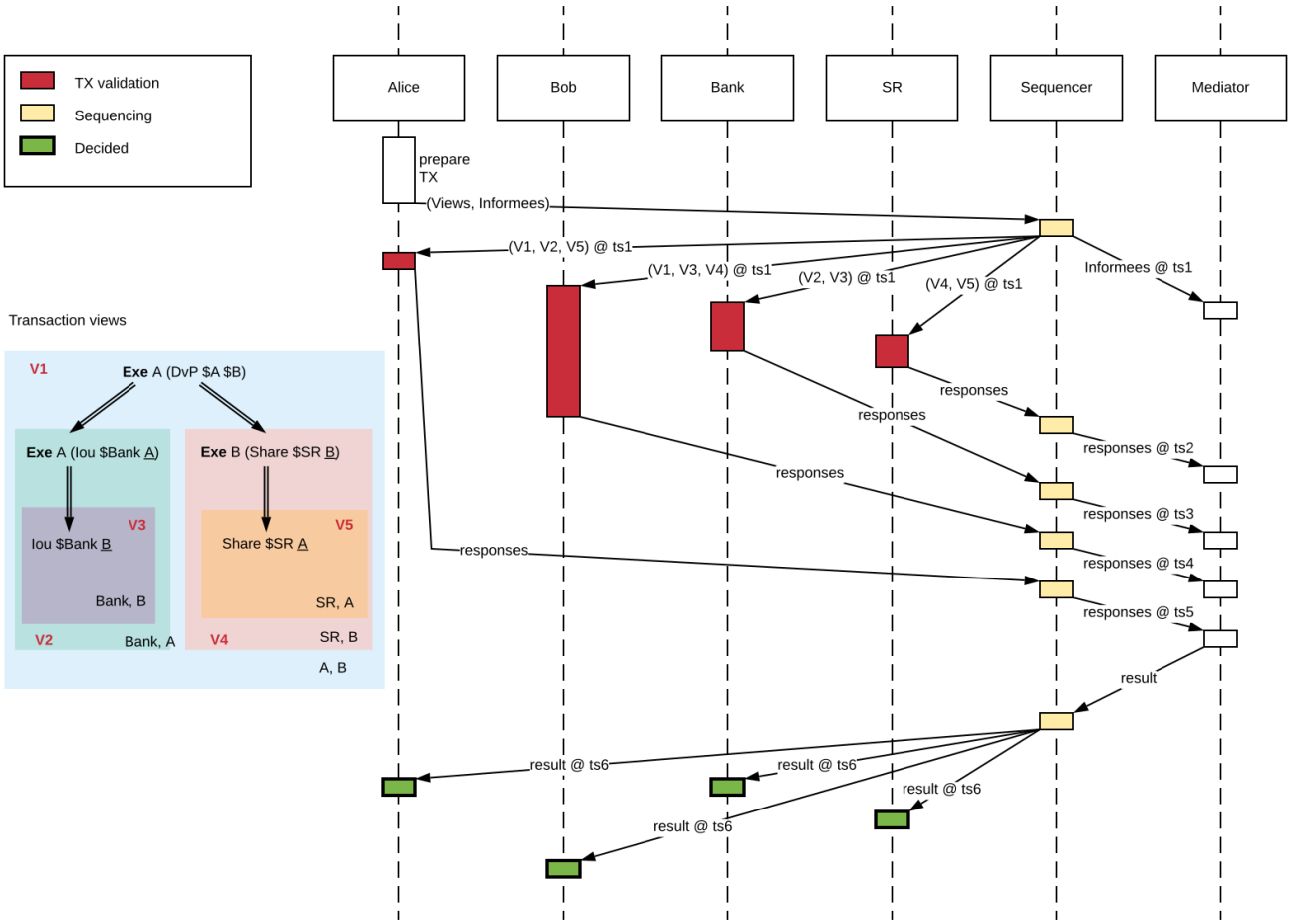
Fig. 3: In the informee tree for the mediator, all transaction contents are blinded.

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 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


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
responses
responses @ ts2
responses @ ts3
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
V2
Bank, A
V5
Share $SR A
SR, A
V4
SR, B
A, B

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*.

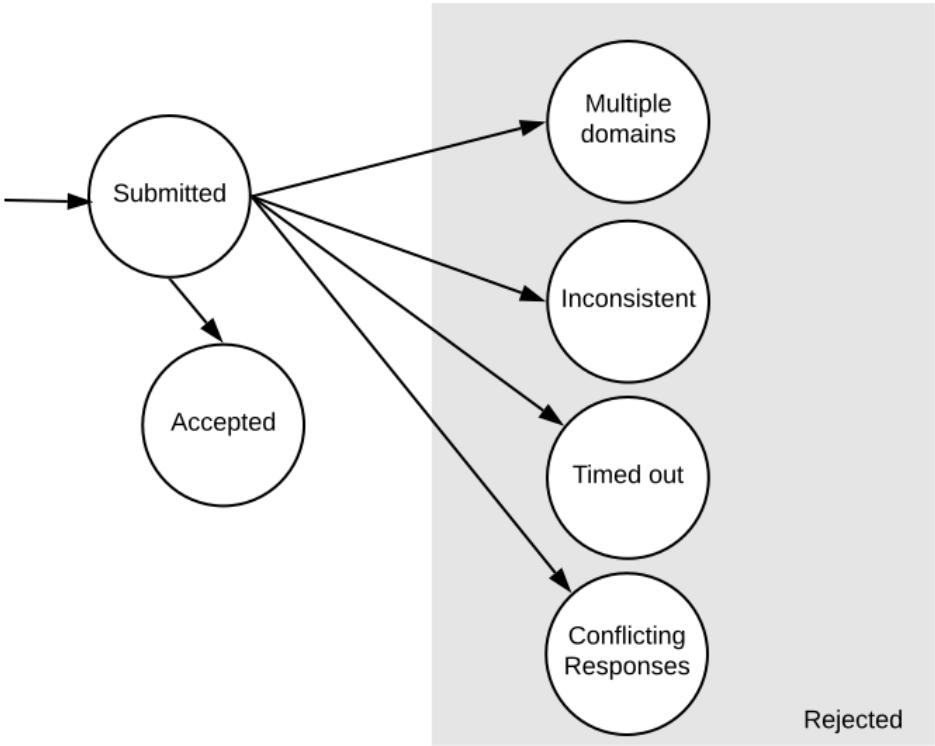
In this document, we focus on the basic version of Canton, with just a single domain. Canton also supports connecting a participant to multiple domains and transferring contracts between domains (see *composability*).

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 *PaintAgreement*) 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*.

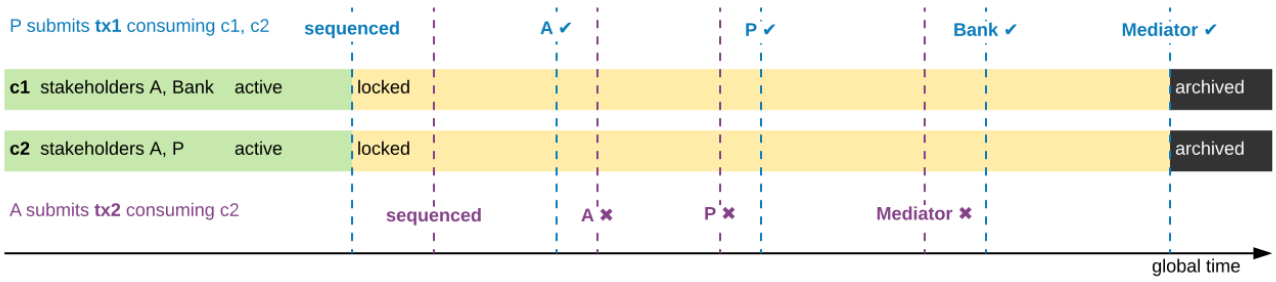


Fig. 4: When two transactions conflict while they are in flight, the later transaction is always rejected.

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

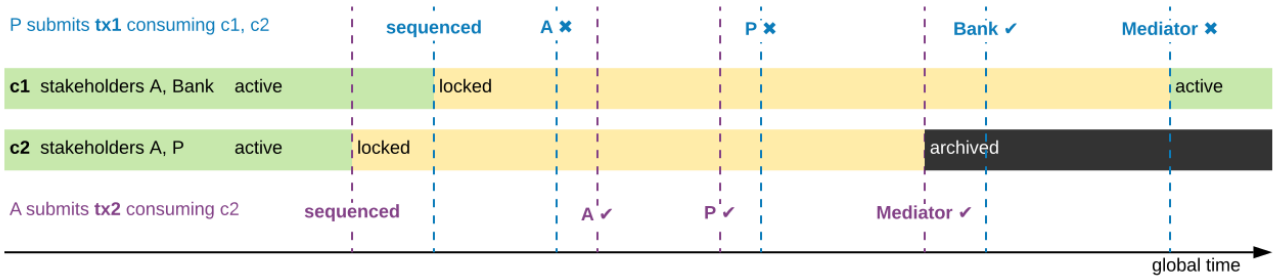


Fig. 5: Transaction *tx2* is now submitted before *tx1*. The consumed contract *c1* remains locked by the rejected transaction until the mediator sends the result message.

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 tries to consume the locked contract *c2*. Later, when *tx1*'s rejection arrives, *c2* becomes active again, but the transaction *tx2* remains rejected.

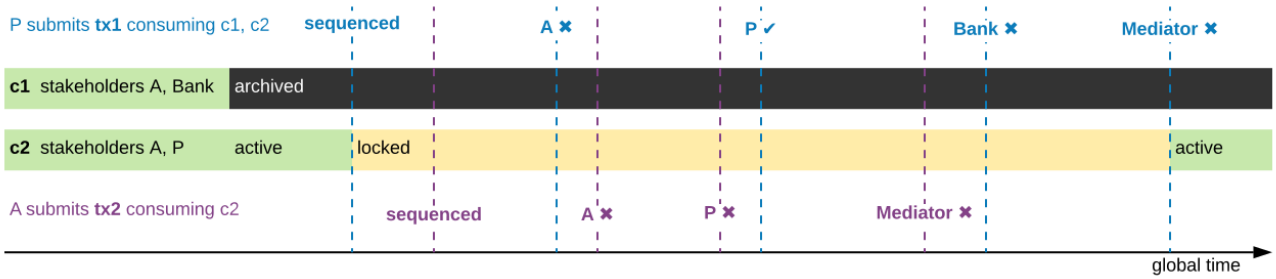


Fig. 6: Even if the earlier transaction *tx1* is rejected later, the later conflicting transaction *tx2* remains rejected and the contract remains locked until the result message.

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 can not 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 can not 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....`

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...`

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 $seed_1$.

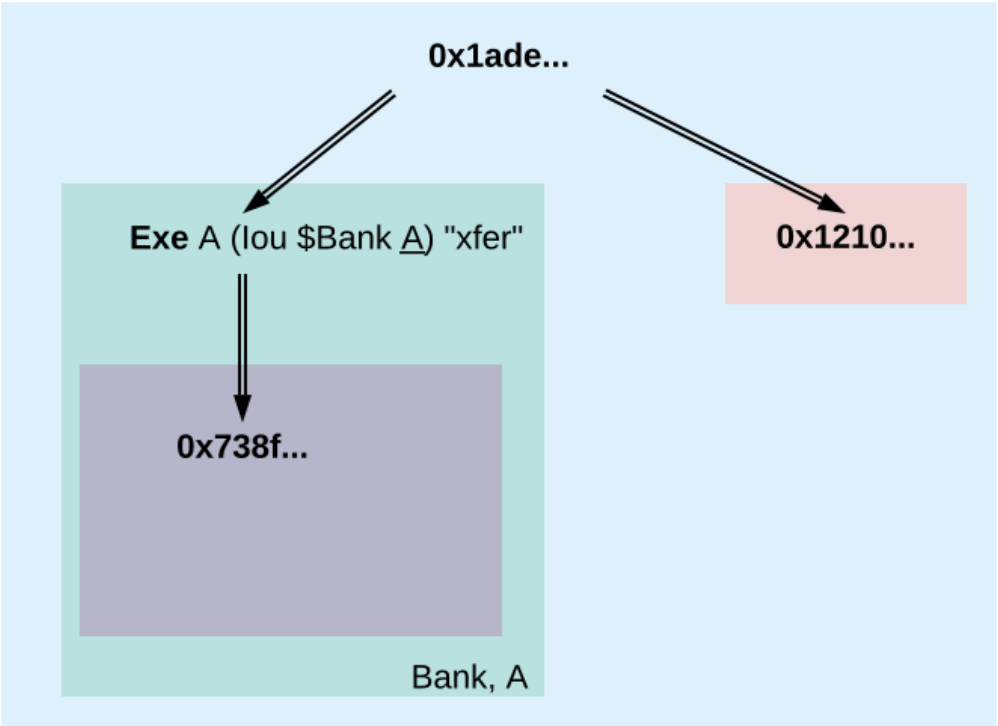


Fig. 7: Idealized Merke tree for the view that exercises the xfer choice on Alice's Iou.

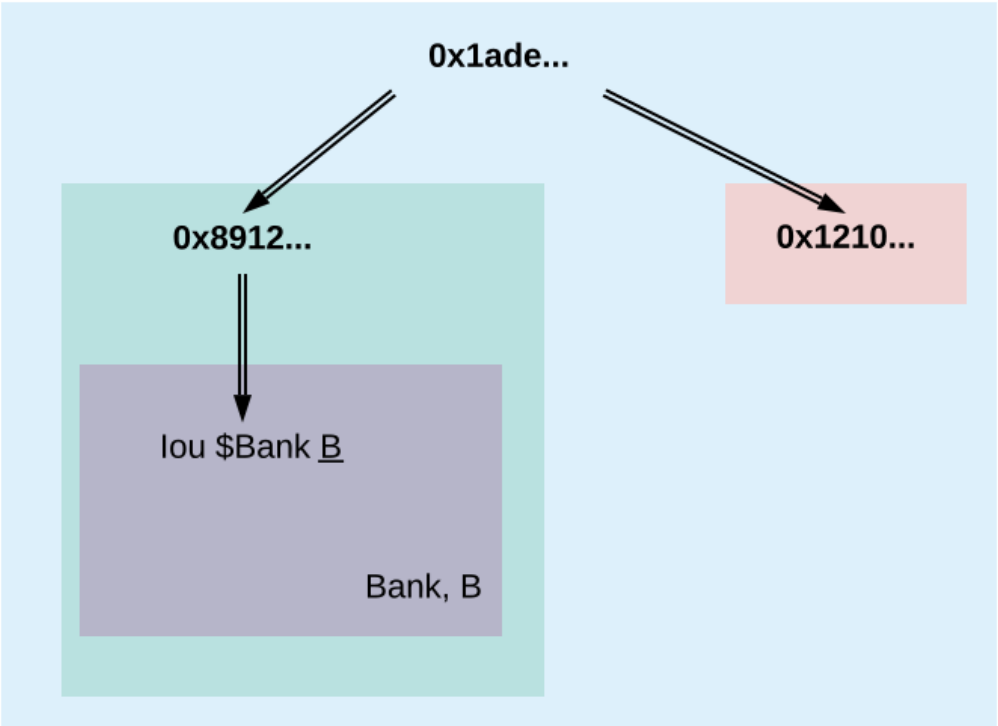


Fig. 8: Idealized Merkle tree for the view that creates Bob's new Iou.

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 witnesses. 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.

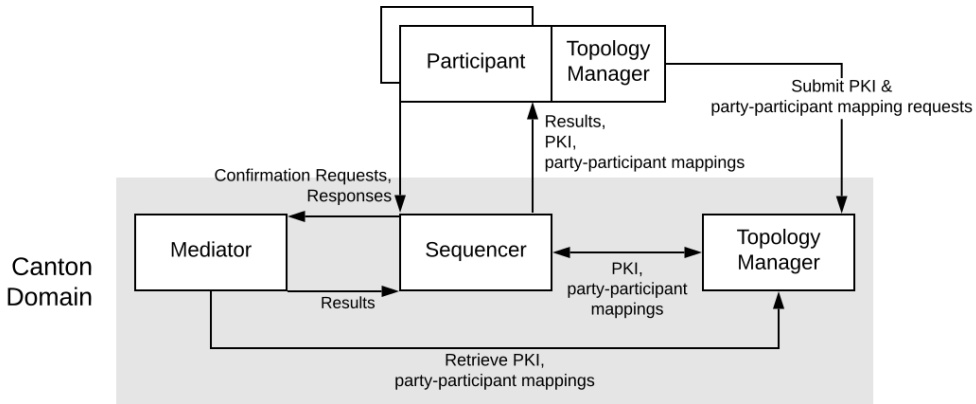
Even though the sequencer persists the encrypted views for a limited period, the domain cannot access the symmetric keys unless it knows the secret key of one of the informee participants. Therefore, the transaction contents remain confidential with respect to the domain.

3.4.2.2 Domain Entities

A Canton domain consists of three entities:

the sequencer
the mediator
and **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.

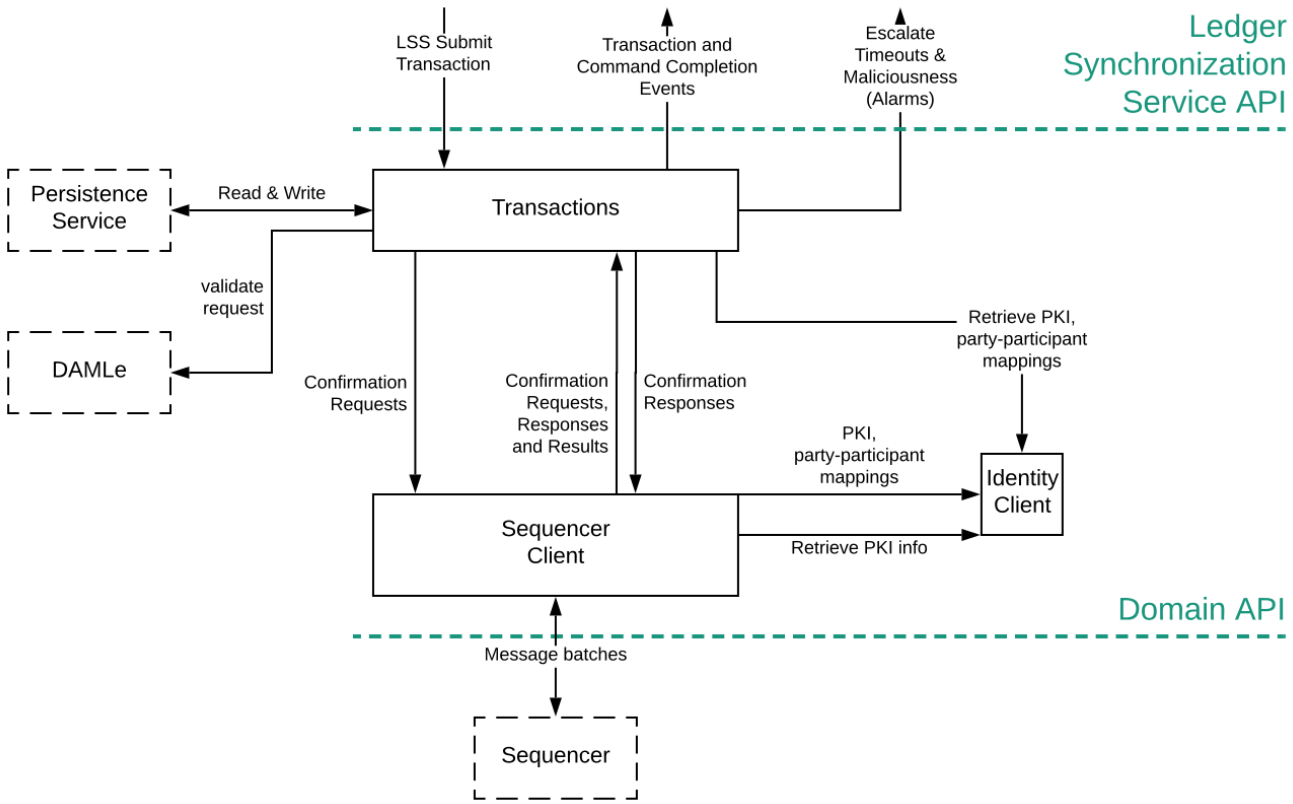
Additionally, for auditability, the mediator persists every received message (containing informee information or confirmation responses) in long term storage and allows an auditor to retrieve messages from this storage.

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.

3.4.2.3 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 further 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).

3.4.2.4 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.

3.4.2.5 Scaling and Performance

Network Scaling

The scaling and performance characteristics of a Canton based system is determined by the scaling characteristics of the entire system and each individual node. Under a Canton based system we understand the set of participant and domain nodes (hereafter, participant or participants and domain or domains respectively), the parties and the deployed Daml workflows. Such a system is not rigid, but flexible, as new participants can connect to multiple domains (preview feature) and new domain nodes and new participant nodes can be added to the network at any point in time. Also participants can be split up by migrating parties off to a new participant node (currently supported as a delicate manual preview feature). Therefore, new computational nodes can be added to the network when needed, allowing the entire system to scale horizontally.

The ledger state in Canton does not exist globally. There is no node that by design hosts all contracts. Instead, participants 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 point for the given input contracts. As an example, if participant Alice and Bank transact on an i-owe-you contract on domain A, another participant Bob or another domain B will not even receive a single bit related to this transaction. This is contrast to Blockchains, where each node has to process each block, independently of how active or directly affected they are by a certain transaction.

A Canton based system can therefore scale horizontally by adding more domains and participants, as long as the workflows are independent of each other. As an example, if there are 100 parties performing multi-lateral transactions with each other, then the system can sharded by running 10 participants with 10 parties each, or say 100 participants with 1 party each. As most of the computation occurs on the participants, a domain can sustain some substantial load from multiple participants, entirely bound by disk and network io of the domain node. But 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 (preview).

If the workflows are made dependent, either by design or by accident, then this scaling property does not hold anymore. Therefore, the modelling of Daml contracts has an impact on the scaling properties and therefore requires some care. As an example, adding an auditor party to each contract in a large solution will require the participant hosting the said party to process every change, and

therefore the vertical scaling limits of an individual node (network and disc io, cpu) will prevent horizontal scaling of the network. This does not mean that workflows must be entirely independent. They can converge and there can be steps that involve coordination across domains (preview), albeit every such (automatic) synchronization comes with additional overhead.

The standard way to recover horizontal scaling is by sharding the workflow across multiple 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, and by adding some intermediate steps for the few cases where the workflows would span across the two shards.

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

Node Scaling

The Canton Enterprise edition 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 behaviour 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 achieve 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.

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, in order to obtain sizing estimates, you must measure the resource usage of dominant workflows using a representative topology and setup of your use-case.

Batching

As every transaction comes with an overhead (signatures, symmetric encryption keys, serialization and wrapping into messages for transport, http headers etc), we recommend to design 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.

Storage Estimation

A priori storage estimation of a Canton installation is tricky. Generally, we can give the following reasoning around the storage used. As explained above, storage usage depends highly on topology, payload, Daml models used and what type of storage layer is configured. However, the following example might be used to 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 can not 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 owner of the Iou are hosted on the same participant whereas the Iou issuer is hosted on a second participant.

In this case, 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 contian 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 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, this makes *11*Y* in total.

Additionally to this, some indexes have to be built by the database in order 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, in order 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 to remove past events and archived contracts based on a timestamp. This way, the storage usage of a Canton deployment can be kept constant by continuously removing obsolete data. Non-repudiation and auditability of the unpruned history is preserved due to the bilateral commitments.

3.4.3 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.

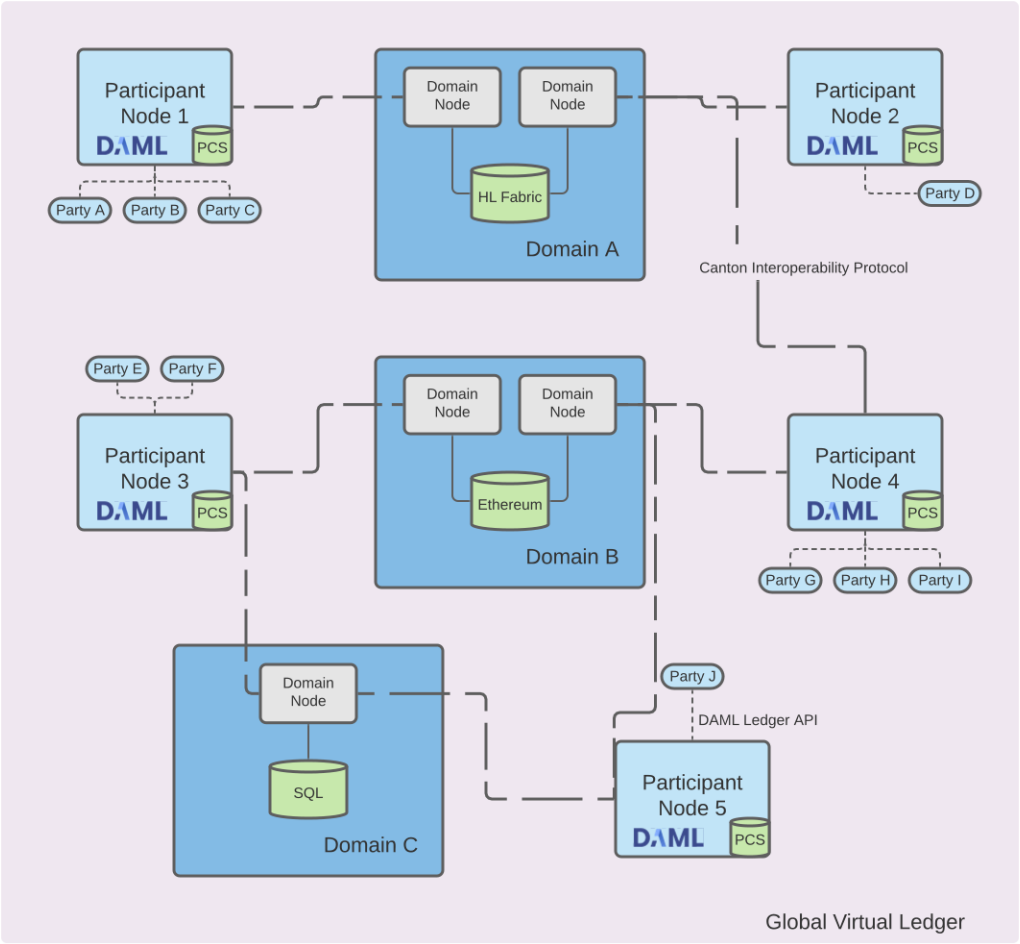
3.4.3.1 Domain-specific Requirements

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.


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

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

Performance

The performance targets cover the entire Canton system and are not broken down to individual component performance targets.

Canton Alpha-level Performance: 5 transactions/second (tps) with up to 3 second latency.
Canton Beta-level Performance: 20 tps with up to 1s latency.

Throughput is measured with a simple Daml workflow where one participant node creates a contract and another participant node must observe the contract. Performance is measured using the Daml Ledger API test tool.

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

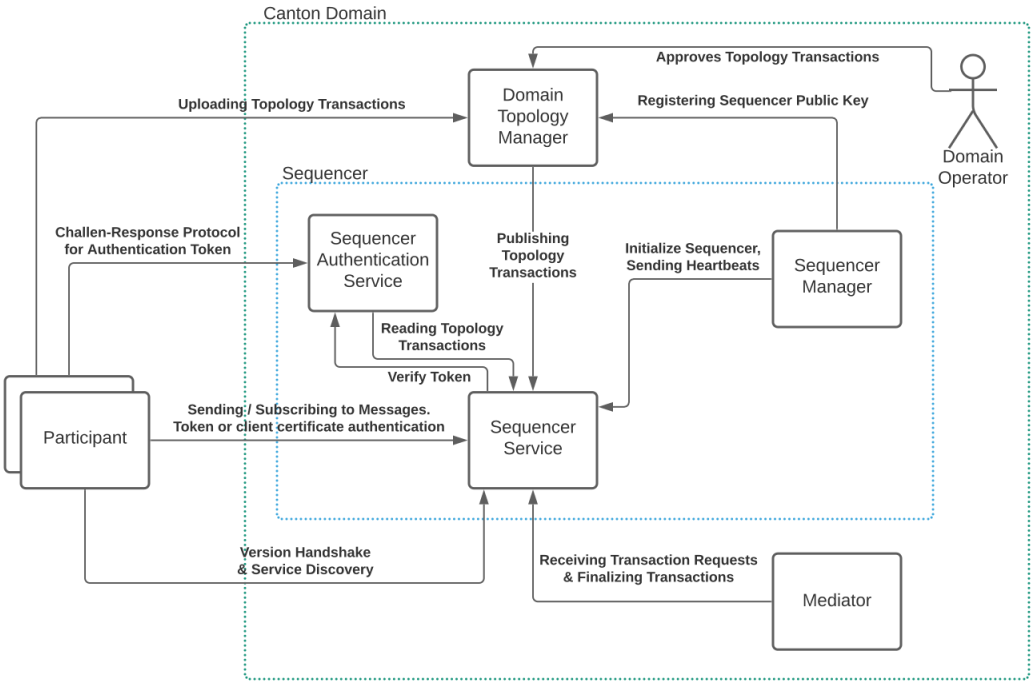
Domain entity compromise recovery: In case of a compromise of a domain entity, the domain must provide procedures to mitigate the impact of the compromise and allow to restore operations.
Standards compliant cryptography: All used cryptographic primitives and their configurations must be compliant to 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.

3.4.3.2 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.

3.4.3.3 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 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 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

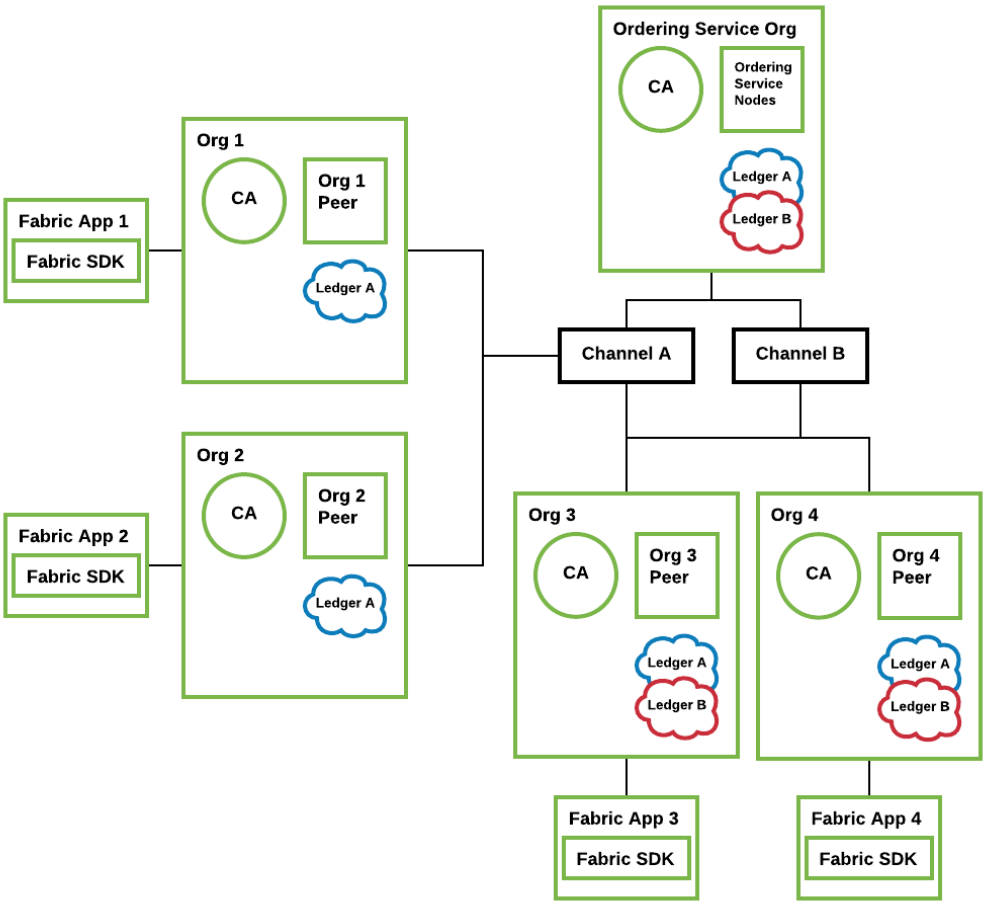
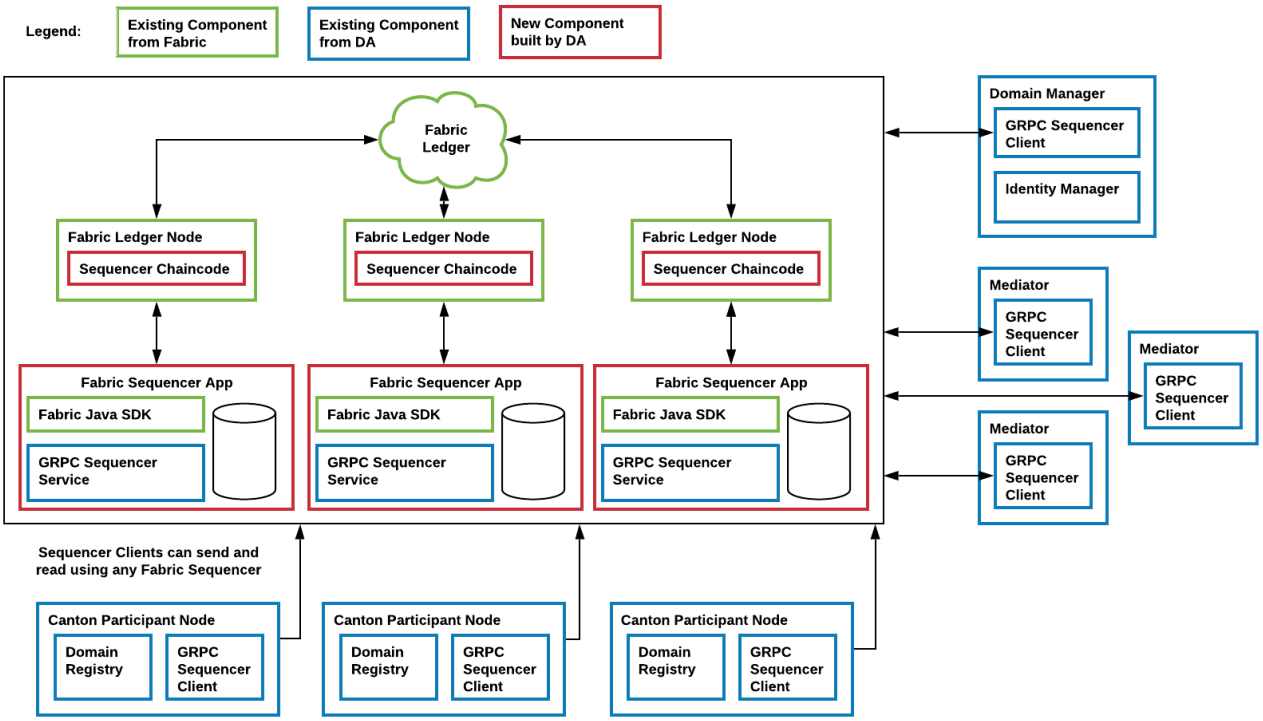


Fig. 9: An example Fabric blockchain network with four organizations. The ordering service has ordering nodes for ordering and distributing blocks on each of the channels defined under the ordering service. Channel A includes all four organizations, while channel B includes only Org 3 and Org 4. Authenticated client applications can send calls to their associated peers on the network.

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.

This paper by IBM Research, India and this article by IBM the many factors that can influence performance.

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.
The mediator is also highly available but the domain manager currently is not.

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 limited resilience against an erroneous participant, for instance it checks that a participant does not send messages to invalid recipients.

Scalability

Horizontal scalability Adding more sequencers to a domain is simply a matter of creating a new organization and a new sequencer application on that organization. It will horizontally scale as well as a Fabric ledger will, which means performance could possibly suffer from a more complex Fabric topology by adding peers and orderer nodes deployed, 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.

3.4.4 High Availability

This section describes how Canton can be run with high availability (HA). Support for HA is being added to the Canton components gradually. The specifics and configuration of HA for each component will be filled in as the implementation is completed. Furthermore, we are mostly starting with cold and warm standby solutions, and will gradually move to hot standby and active-active solutions to improve the mean time to recovery.

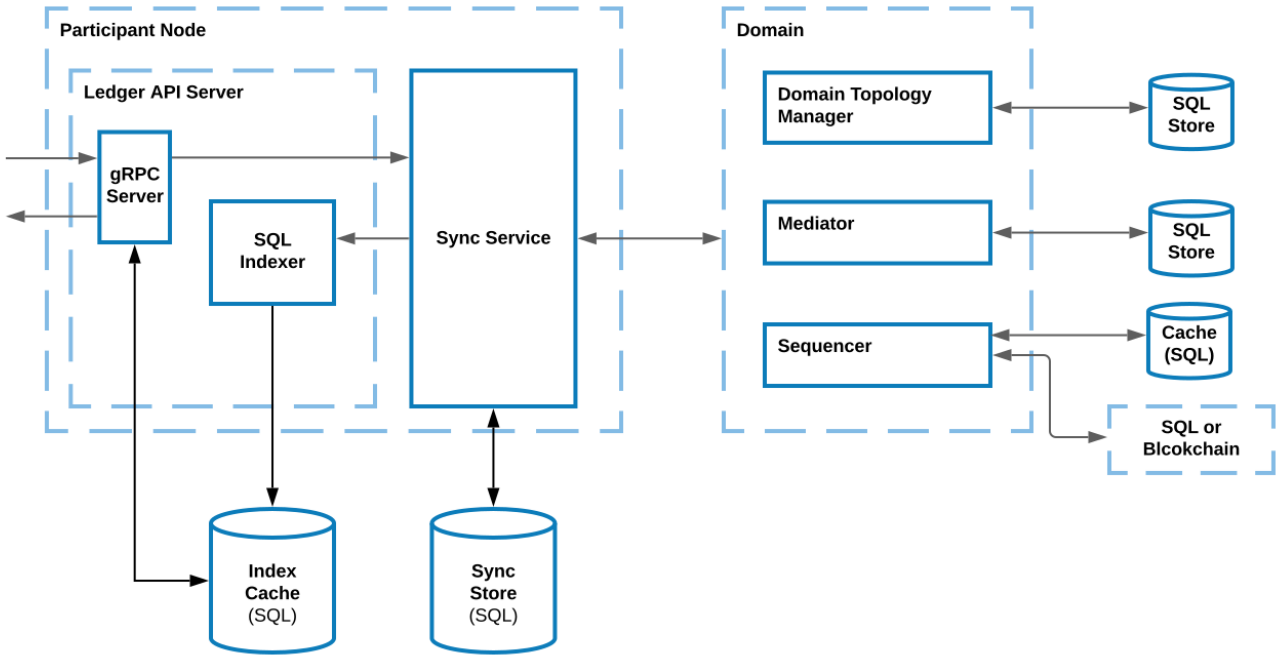
3.4.4.1 Canton High Availability: Overview and Principles

HA of Canton translates into the HA of its main components (see Canton Concepts for a description of each of the components):

the participant nodes, consisting of the following subcomponents:
- gRPC server, which provides the Ledger API access
- sync service, which executes the Canton protocol
- indexer, which builds a read cache

the domains, which have the following subcomponents:
- sequencer, which orders and delivers messages
- mediator, which coordinates transaction processing
- topology manager, that manages known identities on the domain
- domain service, which manages registration

The components, their subcomponents, and their data stores are illustrated in the image below, where the arrows show the direction of the data flow, dashed lines denote the logical components, and solid lines denote maximal process separation, i.e., the subcomponents which can be run in separate processes.



While multiple components can be run in the same process, to achieve HA, you should run each component in its own process. That way, the availability of one component is not affected by the lack of availability of other ones, except for the workflows that directly involve both components.

In particular, the availability of a Canton participant *A* is not affected by the availability of a participant *B*, except for the workflows that:

1. involve both *A* and *B* and where
2. *A* and *B* don't have the same visibility into workflow data, i.e., they manage different parties involved in the workflow.

That is, if *A* and *B* host the same party *P*, then processing of transactions involving *P* can still continue as long as either *A* or *B* is available. However, note that application operating on behalf of *P* currently cannot transparently fail over from *A* to *B* or vice versa, due to the difference in offsets emitted on each participant.

Furthermore, the availability of *A* is also not affected by the availability of the domain *D*, except for the workflows that use *D*. This allows each participant and domain to take care of its HA separately. To achieve HA, the components will be replicated, and all replicas of the same component are assumed to have the same trust assumptions, i.e., the operators of one replica must trust the operators of the other replicas.

In general, whenever a component is backed by a database/ledger, the HA of the component currently relies on the HA of the database/ledger. The component's operator must handle the HA of the database separately. All database-backed Canton components are designed to be tolerant to temporary database outages. During the failover period for the database, such Canton components halt processing until the database becomes available again, and resume processing thereafter. Transactions that involve these Canton components may time out if the failover takes too long. Nevertheless, they can be safely resubmitted, as command deduplication provides idempotency.

Canton components can expose a *health endpoint*, that can be used to check the health of the components and its subcomponents.

In the following sections, we describe the HA approach of each component.

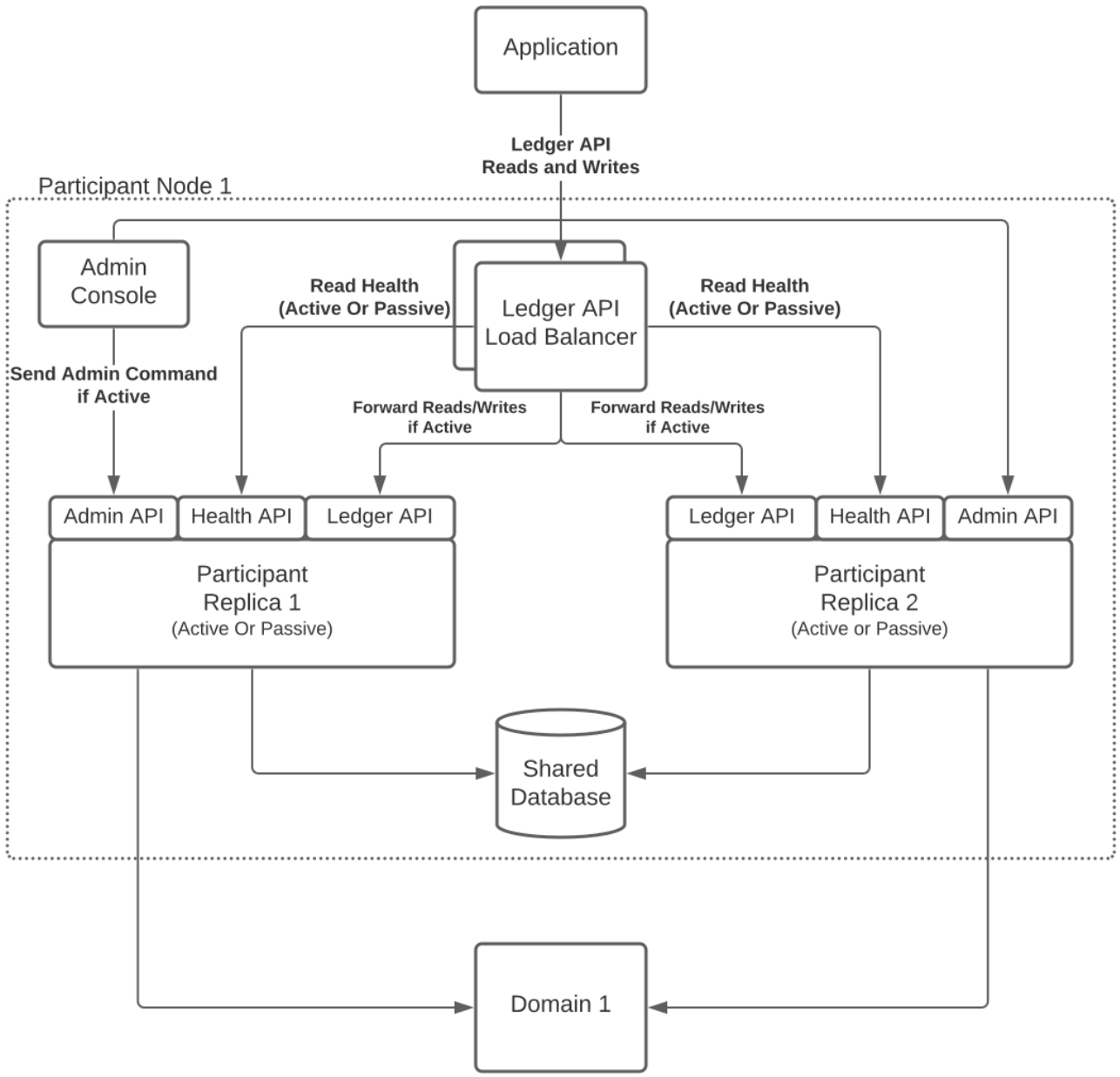
Important: This feature is only available in Canton Enterprise

3.4.4.2 Replicated Participant Node Architecture

High availability of a participant node is achieved with a replicated participant node in an active-passive configuration, where the active replica is serving requests and one or more passive replicas are in a warm stand-by mode ready to take over when the active replica fails.

High-Level System Design

A participant node consists of multiple participant node replicas using a shared database and each replica exposing its own ledger API. However from an application point of view, the fact that multiple replicas exists can be hidden by exposing a single ledger API endpoint through a highly available load balancer.



Why a Shared Database?

The replicas of a replicated participant node share the same database, which is required for two reasons:

> Share the command ID deduplication state of the ledger API command submission service between replicas to prevent double submission of commands in case of fail-over.
> Obtain consistent ledger offsets across the replicas, otherwise the application could not seamlessly fail-over to another replica. The ledger offsets are decided by the database based on the insertion order of publishing events in the multi-domain event log, i.e., the ledger offset derivation is not deterministic.

Participant Node Replica Monitoring and Fail-Over

Operating a participant node in a replicated active-passive configuration with a shared database requires to establish the active replica, i.e., perform a leader election, and to enforce a single writer, i.e., the active replica, to the shared database.

We are using exclusive application-level database locks tied to the lifetime of the connection to the database to achieve leader election and a enforce single writer. Alternative existing approaches for leader election, such as using Raft, are not suitable because in between the leader check and the use of the shared resource, i.e., writing to the database, the leader status could have been lost and we cannot guarantee a single writer.

Leader Election through Exclusive Lock Acquisition

A participant node replica tries to acquire an exclusive application level lock (e.g. Postgres advisory lock) bound to a particular database connection and use that same connection for all writes that are not idempotent. The replica that has acquired the lock is the leader and the active replica. Using the same connection for writes ensures that the lock is held while writes are performed.

Lock ID Allocation

The exclusive application level locks are identified by a 30bit integer. The lock id is allocated based on the scope name of the lock and a lock counter. The lock counter differentiates locks used in Canton from each other, depending on their usage. The scope ensures the uniqueness of the lock id for a given lock counter. For the allocation the scope and counter are hashed and truncated to 30bit to generate a unique lock id.

On Oracle the lock scope is the schema name, i.e., the user name. On Postgres it is the name of the database. The participant replicas must allocate the same lock ids for the same lock counter, therefore it is crucial that the replicas are configured with the same storage configuration, e.g., for Oracle using the same username to allocate the lock ids with the same scope.

Enforce Passive Replica

The replicas that do not hold the exclusive lock are passive and cannot write to the shared database. To avoid any attempts to write to the database, which would fail and produce an error, we use a coarse-grained guard on domain connectivity and API services to enforce a passive replica.

To prevent the passive replica from processing any domain events and reject incoming ledger API requests, we keep the passive replica disconnected from the domains as a coarse-grained enforcement.

Lock Loss and Fail-Over

If the active replica crashes or loses connection to the database, the lock will be released and a passive replica can claim the lock and become active. Any pending writes in the formerly active replica will fail as the underlying connection and the corresponding lock has been lost.

There is a grace period for the active replica to rebuild the connection and reclaim the lock to avoid unnecessary fail-overs on short connection interruptions. The passive replicas continuously try to acquire the lock with a configurable interval. Once the lock is acquired, the participant replication manager sets the state of the replica to active and completes the fail-over.

As part of a passive replica becoming active, the replica is connected to previously connected domains to resume processing of events. Further the new active replica now accepts incoming requests, e.g., on the ledger API. On the other hand, the former active replica that is now passive needs to reject any incoming requests as the replica can no longer write to the shared database.

Ledger API Client Fail-Over via Load Balancer

To hide the fact that a participant is replicated and to offer a single ledger API endpoint towards applications, we recommend the usage of layer 4 (=TCP level), highly available load balancer.

The load balancer (LB) is configured with a pool of backend servers based on the ledger API server addresses and ports of the participant node replicas. The participant node replicas expose their status if they are the active or passive replica via a health endpoint. The LB periodically checks the health API endpoint of the replicas and marks a backend server offline if the replica is passive. Thus the load balancer only sends requests to the active backend server. The polling frequency of the health endpoints affect the fail-over times.

During fail-over requests may still be send to the former active replica, which will be rejected and the application has to retry the submission of commands in that case until they are forwarded to the new active replica.

3.4.4.3 Domain HA

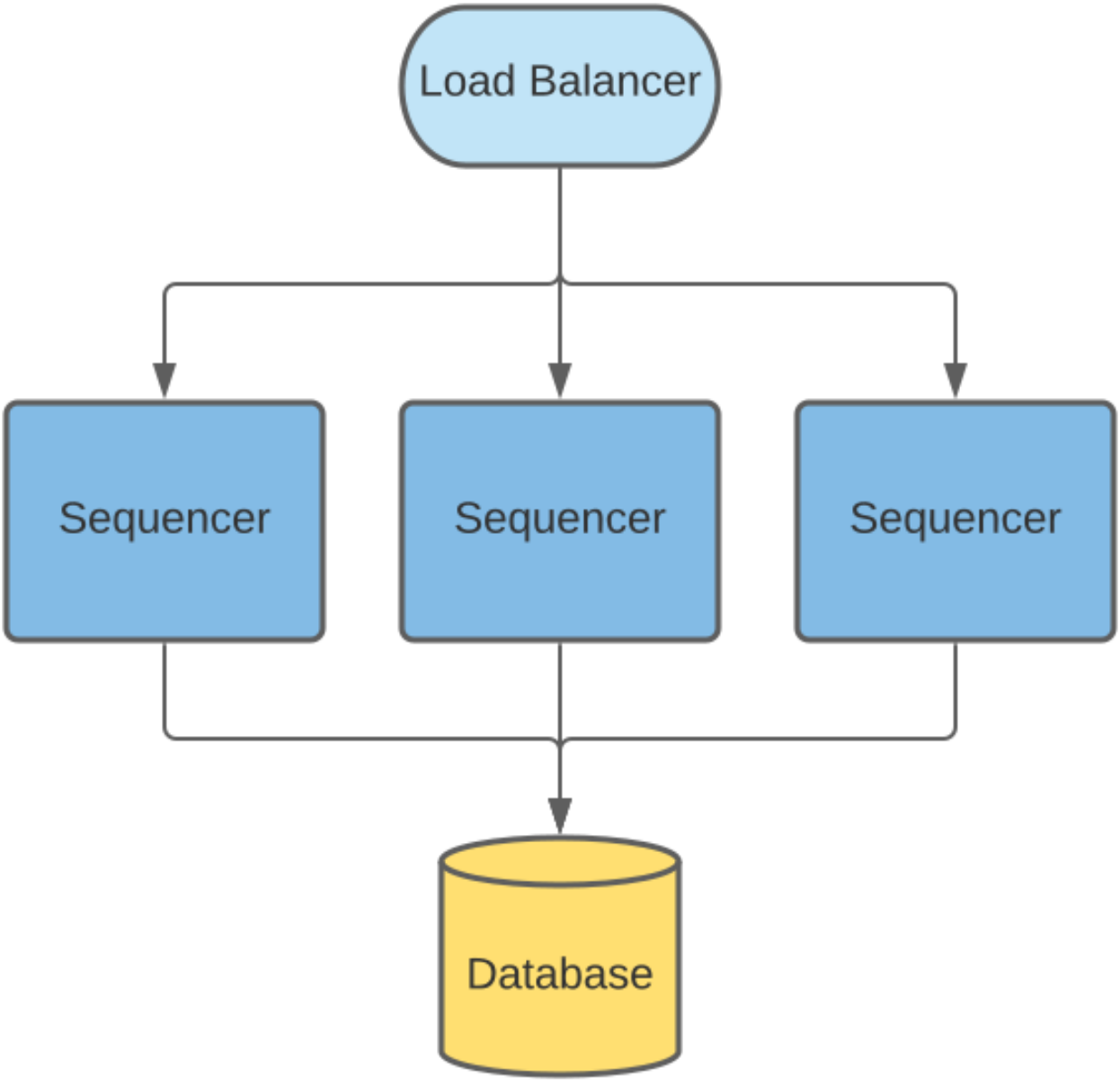
A domain is fully available only when all of its subcomponents are available. However, transaction processing can still run over the domain even if only the mediator and the sequencer are available. The domain services handle new connections to domains, and the topology manager handles the changes to the topology state; unavailability of these two components affects only the services they handle. As all of these components can run in separate processes, we handle the HA of each component separately.

Sequencer HA

The HA properties of the Sequencer depend on the chosen implementation. When the sequencer is based on a HA ledger, such as Hyperledger Fabric, the sequencer automatically becomes HA. The domain service can return multiple sequenced endpoints, any of which can be used to interact with the underlying ledger.

For the database sequencer, we use an active-active setup over a shared database. The setup relies on the database for both HA and consistency.

Database Sequencer HA



The database Sequencer uses the database itself to ensure that events are sequenced with a consistent order. Many Sequencer nodes can be deployed where each node has a Sequencer reader and writer component, all of these components can concurrently read and write to the same database. A load balancer can be used to evenly distribute requests between these nodes. The canton health endpoint can be used to halt sending requests to a node that reports itself as unhealthy.

Sequencers nodes are statically configured with the total number of possible Sequencer nodes and each node is assigned a distinct index from this range. This index is used to partition available event timestamps to ensure two sequencer node will never use the same event id/timestamp.

Events are written to the `events` table and can be read in ascending timestamp order. To provide a continuous monotonic stream of events, readers need to know the point at which events can be read without the risk of an earlier event being inserted by a writer process. To do this writers regularly update a `watermark` table where they publish their latest event timestamp. Readers take the

minimum timestamp from this table as the point they can safely query events for.

If a Sequencer node was to fail, it would stop updating its `watermark` value and when it becomes the minimum timestamp this will cause all readers to effectively pause at this point (at they cannot read beyond this point). Other Sequencers writers when updating their own watermark also check that the other sequencer watermarks are being updated in a timely manner. If it is noticed that a Sequencer node has not updated its watermark within a configurable interval then it will be marked as offline and this watermark will no longer be included in the query for the minimum event timestamp. This causes future events from the offline Sequencer to be ignored after this timestamp. For this process to operate optimally the clocks of the hosts of the Sequencer nodes are expected to be synchronized - this is considered reasonable for where all Sequencer hosts are co-located and NTP is used.

If the failed Sequencer has recovered and would like to resume operation, it should delete all events past its last know watermark to avoid incorrectly re-inserting them into the events the readers will see, as readers may have read subsequent events by this time. This is safe to do without effecting events that have been read as any events written by the offline Sequencer after it is marked offline are ignored by readers. It should then replace its old watermark with a new timestamp for events it will start inserting then resume normal operation, ensuring that this is greater than any existing value.

When a Sequencer fails and resumes operation there will be short pause in reading from other Sequencers due to updates to the watermark table. However requests to the other Sequencer nodes should continue successfully, and any events written during this period will be available to read as soon as the pause has completed. Any send requests that were being processed by the failed Sequencer process will likely be lost, but can be safely retried once their max-sequencing-time has been exceeded without the risk of creating duplicate events.

Mediator HA

The approach for mediator node HA follows the same principles as outlined for participant HA in *Replicated Participant Node Architecture*. Namely a mediator node is replicated and only one replica is active. All replicas of the same mediator node share the same database, both for sharing the state as well as to coordinate the active mediator node replica.

3.4.5 Identity Management

3.4.5.1 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 requirements on *identity provider integration* and *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 can not 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.

3.4.5.2 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 can not 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 can not 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 will introduce an 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 will 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 will 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 will 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 can not 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
  override def pretty: Pretty[Namespace] = prettyOfParam(_.fingerprint)
```

(continues on next page)

(continued from previous page)

```
}

/** 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 will 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 an topology transaction $\{.\}$) depend on the command ω. Most but not all transactions require the signatures to be nested in some form. Generally, we note that 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
    with HasProtoV0[v0.NamespaceDelegation] {
```

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
    with HasProtoV0[v0.IdentifierDelegation] {
```

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 an 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 authorised 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
    with HasProtoV0[v0.OwnerToKeyMapping] {
```

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
    with HasProtoV0[v0.PartyToParticipant] {
```

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 can not be involved in any transaction, a means participant is *active*, c means participant can not 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
    with HasProtoV0[v0.ParticipantState] {

  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. Therefore, the distinction between *submission* and *confirmation* permissions in the party to participant mappings are only respected by the default implementation. A malicious submitter with *confirmation* permissions 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.

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

$$\begin{aligned} S_t &= S_{t-1} + \{.,\omega\}_t^{s_k,s_{k'}} \\ &= \bigoplus_{t'<t} \{.,\omega\}_{t'}^{[s_.]} \\ &= [(.,I) \Rightarrow p] + [((X,I) \rightarrow Y)] + [(.,N)] \end{aligned}$$

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 an 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.

Replay Attacks

The topology transactions published on a domain D are not pinned to the particular domain. Restricting topology transactions to specific domains would prevent a replay, but would also create an administrative burden. A malicious domain might pretend a presence of a participant on a domain. However, impersonating that participant or domain entity is impossible without the secret private key which means that a replay of topology transactions has no impact on integrity or privacy.

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 $X \rightarrow Y$ to all participants, we broadcast $T = (H(X), t_id)$ 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 $N \rightarrow K$ we note that by only broadcasting $H(N)$ instead of N, other participants can not 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 hold 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: $\{(P, I_k) \rightarrow (N_2, I_k)\}^{s_k}$

A straight-forward extension to permission a party on a second participant in another identity namespace is: $\{(P, I_k) \rightarrow (N_2, I_l)\}^{s_k, s_l}$ 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 cooperate.

First, we note that we avoid collision problems by using globally unique identifiers derived from namespaces generated by root keys by design.

Second, we note that we do not need to have complete consistency of identities between the committers. All that is required is a sufficient overlap.

We first introduce a new mapping transaction denoted the transfer permission as $\{P \rightarrow D_T\}$ 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 $\{P \rightarrow D\}$ 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-our on D_S will be rejected if $(P \rightarrow [N])^{t_1}_{D_S} \cap (P \rightarrow [N])^{t_0}_{D_T} = \emptyset$

2) transfer-in on D_T will be rejected if $(P \rightarrow [N])^{t_1}_{D_S} \cap (P \rightarrow [N])^{t_2}_{D_T} = \emptyset$

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 $\{+, (N, I_k) \to p_2\}^{\tilde{s}_k}\{-, (N, I_k) \to p_1\}^{\tilde{s}_k}$

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?*

$\{+, P \to N_1\}\{+, \to N_2\}$

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.*

$$\{+, (P, I_k) \to (N_2, I_l)\}^{I_k, I_l}\{-, P \to (N_2, I_k)\}^{I_k}$$

3.4.5.3 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): IdentityProvidingServiceClient = {
    domains += (domainClient.domainId -> domainClient)
```

(continues on next page)

(continued from previous page)

```
    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] {

  /** 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.
    */
  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

  /** 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.
    *
```

(continues on next page)

(continued from previous page)

```
    * 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
    */
  def snapshot(timestamp: CantonTimestamp)(implicit traceContext: TraceContext):
↪Future[T]

  /** Waits until a snapshot is available */
  def awaitSnapshot(timestamp: CantonTimestamp)(implicit traceContext:
↪TraceContext): Future[T]

  /** Returns the topology information at a certain point in time
    *
    * Fails with an exception if the state is not yet known.
    */
  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]]

}

/** The client that provides the topology information on a per domain basis
  */
trait DomainTopologyClient extends TopologyClientApi[TopologySnapshot] with
↪AutoCloseable {

  /** Subscribe to topology information updates */
  def subscribe(subscriber: DomainTopologyClient.Subscriber): Unit
```

(continues on next page)

(continued from previous page)

```
  /** Remove observer from topology information updates */
  def unsubscribe(subscriber: DomainTopologyClient.Subscriber): Unit

  /** Wait for a condition to become true according to the current snapshot
↪approximation
    *
    * @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]

  /** List all the dynamic domain parameters (past and current) */
  def listDynamicDomainParametersChanges()(implicit
      traceContext: TraceContext
  ): Future[Seq[DynamicDomainParameters.WithValidity]]
}

object DomainTopologyClient {

  trait Subscriber {

    /** Inform the subscriber about the processed transactions */
    def observed(
        sequencedTimestamp: SequencedTime,
        effectiveTimestamp: EffectiveTime,
        sequencerCounter: SequencerCounter,
        transactions: Seq[SignedTopologyTransaction[TopologyChangeOp]],
    )(implicit traceContext: TraceContext): Unit

  }

  trait TransactionSubscriber extends Subscriber {

    def observedTransaction(transaction:
↪SignedTopologyTransaction[TopologyChangeOp])(implicit
        traceContext: TraceContext
    ): Unit

    final override def observed(
        sequencedTimestamp: SequencedTime,
        effectiveTimestamp: EffectiveTime,
        sequencerCounter: SequencerCounter,
        transactions: Seq[SignedTopologyTransaction[TopologyChangeOp]],
    )(implicit traceContext: TraceContext): Unit =
      transactions.foreach(observedTransaction)

  }

}

trait BaseTopologySnapshotClient {

  protected implicit def executionContext: ExecutionContext
```

(continues on next page)

(continued from previous page)

```
  /** 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]]]

  /** 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 true if there is at least one participant that can confirm */
  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]
```

(continues on next page)

(continued from previous page)

```
  /** 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]]

  /** Returns a list of all known parties on this domain */
  def inspectKnownParties(
      filterParty: String,
      filterParticipant: String,
      limit: Int,
  ): Future[Map[PartyId, Map[ParticipantId, ParticipantAttributes]]]

}

/** 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 signing keys */
  def encryptionKeys(owner: KeyOwner): Future[Seq[EncryptionPublicKey]]

  /** Returns a list of all known parties on this domain */
  def inspectKeys(
      filterOwner: String,
      filterOwnerType: Option[KeyOwnerCode],
      limit: Int,
  ): Future[Map[KeyOwner, KeyCollection]]

}

/** The subset of the topology client, providing participant state information */
trait ParticipantTopologySnapshotClient {
```

(continues on next page)

(continued from previous page)

```
  this: BaseTopologySnapshotClient =>

  // used by domain to fetch all participants
  // used by participant to know to which participant to send a use package
↪contract (will be removed)
  @Deprecated
  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 */
  def mediators(): Future[Seq[MediatorId]]

  def isMediatorActive(mediatorId: MediatorId): Future[Boolean] =
    mediators().map(_.contains(mediatorId))
}

trait CertificateSnapshotClient {

  this: BaseTopologySnapshotClient =>

  def hasParticipantCertificate(participantId: ParticipantId)(implicit
      traceContext: TraceContext
  ): Future[Boolean] =
    findParticipantCertificate(participantId).map(_.isDefined)

  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
    * @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,
```

(continues on next page)

(continued from previous page)

```
      packages: Set[PackageId],
  ): EitherT[Future, PackageId, Set[PackageId]]

}

trait DomainGovernanceSnapshotClient {
  this: BaseTopologySnapshotClient =>

  def findDynamicDomainParametersOrDefault(warnOnUsingDefault: Boolean =
↪true)(implicit
      traceContext: TraceContext
  ): Future[DynamicDomainParameters]

}

trait TopologySnapshot
    extends PartyTopologySnapshotClient
    with BaseTopologySnapshotClient
    with ParticipantTopologySnapshotClient
    with KeyTopologySnapshotClient
    with CertificateSnapshotClient
    with VettedPackagesSnapshotClient
    with MediatorDomainStateClient
    with DomainGovernanceSnapshotClient {}
```

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

  /** 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 encryption key */
  def decrypt[M](encryptedMessage: Encrypted[M])(
      deserialize: ByteString => Either[DeserializationError, M]
  ): 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)

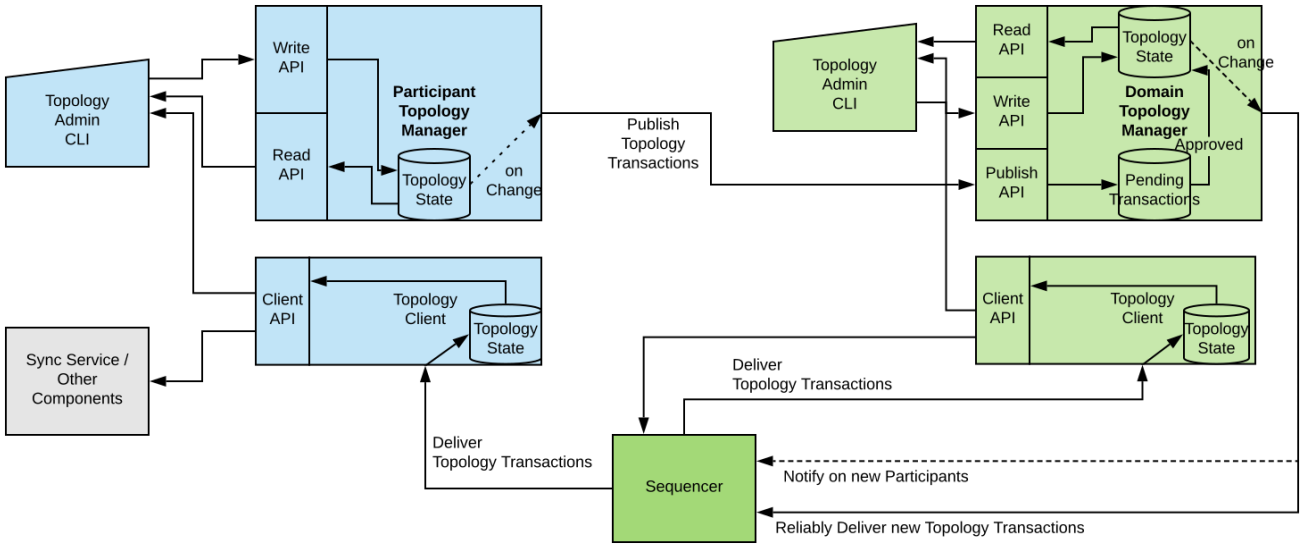
```
    */
  def verifySignature(
      hash: Hash,
      signer: KeyOwner,
      signature: Signature,
  ): EitherT[Future, SignatureCheckError, Unit]

  /** 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,
  ): EitherT[Future, SyncCryptoError, Encrypted[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.

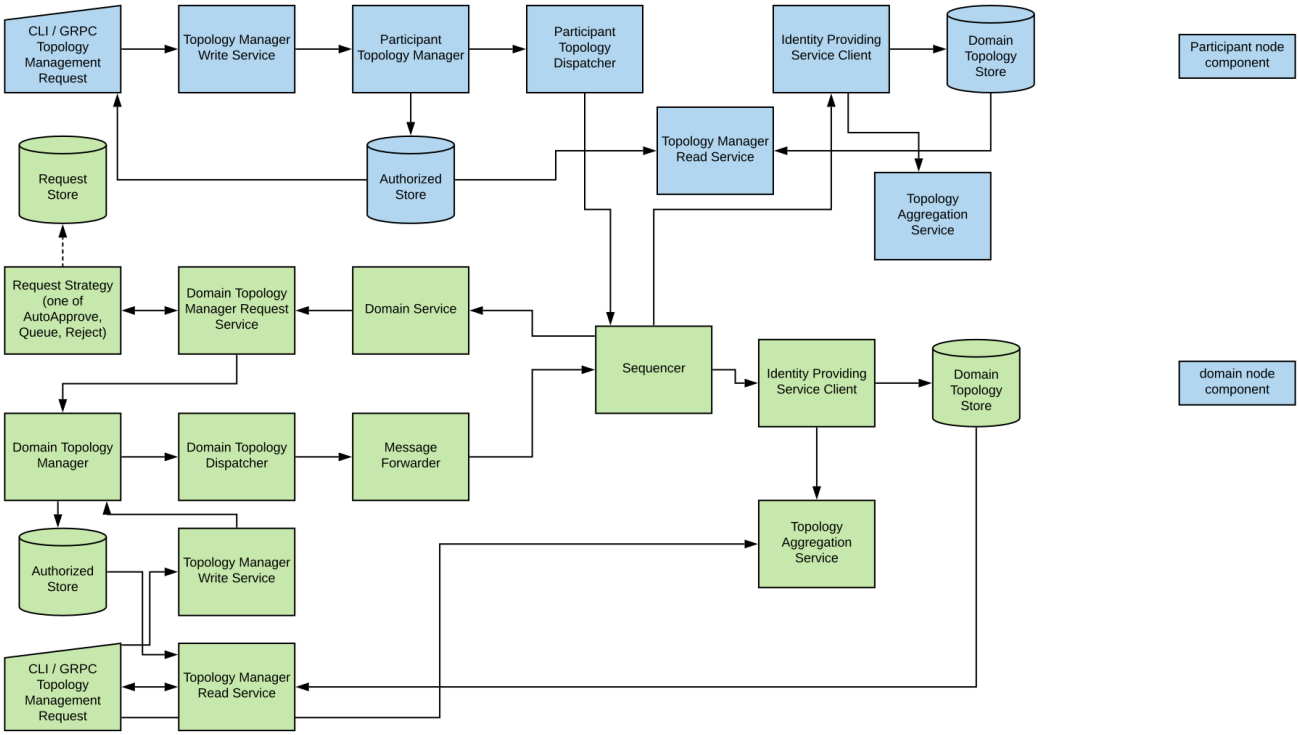
High-Level Picture

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 effect changes to the topology state, an administrator needs to create a new topology transaction and authorize is 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 by either importing authorized transactions or create new authorized transactions himself. 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 an 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 the added functionality required for a domain, allowing 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 identity store is the store of topology transactions that have been timestamped by the sequencer. The authorized store of a domain and the domain identity store will 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 identity 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 services just serves inspection purposes in order to look deeply into the topology state. The read services plugs directly onto a topology store and expose the content via GRPC.

3.4.6 Research Publications

Canton and the underlying theory is described in the following research publications:

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.
CantonCoin: Gaining Horizontal Scalability and Privacy with Distributed Commits Instead of Global Consensus explains how Canton's distributed commit protocol enables a scalable and privacy-preserving coin design.
Abstract: Bitcoin sought to create a trustless and censorship-resistant paymentsystem. However, to obtain a practical day-to-day payment systemcapable of serving the global needs, these goals must be balanced with many other properties, such as scalability and privacy. In this paper, we show that replacing state machine replication (the mechanism underlying Bitcoin and most other blockchains) with a distributed commit protocol over private local ledgers removes the scalability bottlenecks and enhances privacy. We demonstrate our approach usingthe Canton commit protocol, resulting in the CantonCoin prototype cryptocurrency.
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.

3.4.7 Security

This section covers many security related aspects of Canton.

Note: This section is in development and will be extended over the coming months.

3.4.7.1 Ledger-API Authorization

The Ledger Api provides *authorization support* using JWT tokens. While the JWT token authorization allows to authorize third party applications 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 on each startup a new admin bearer token (64 bytes, randomly generated, hex-encoded) 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.

Chapter 4

Help

4.1 Troubleshooting

4.1.1 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.

4.1.2 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.
```

4.1.3 Modeling questions

4.1.3.1 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.

4.1.3.2 How to model rights

Use a contract choice to model a right. A party exercises that right by exercising the choice.

4.1.3.3 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.

4.1.3.4 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.

4.1.3.5 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.

4.1.3.6 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.

4.1.3.7 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.

4.1.4 Testing questions

4.1.4.1 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
```

4.1.4.2 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.

4.2 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.

4.2.1 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.

If you need private support, or want consultation from Digital Asset about how to build your Daml application, they offer paid support. Please contact Digital Asset to ask about pricing.

4.3 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.

4.3.1 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.

4.3.1.1 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
2.0	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

4.3.1.2 Summary of Ledger API Changes

Ledger API Version	Changes
2.0	Introduce User Management Service Introduce Metering Report Service Remove Reset Service Deprecate Ledger Identity Service Make ledger_id and application_id fields optional Change error codes returned by the gRPC services
1.12	Introduce Daml-LF 1.14
1.11	Introduce Daml-LF 1.13
1.10	Introduce Daml-LF 1.12 Stabilize *participant pruning*
1.9	Introduce Daml-LF 1.11
1.8	Introduce Multi-Party Submissions
<= 1.7	See Daml (SDK) release notes of same version number.

4.3.2 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.

4.3.3 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.

4.3.4 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.

Chapter 5

Reference

5.1 Glossary of concepts

5.1.1 Key Concepts

5.1.1.1 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.

5.1.1.2 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.

5.1.1.3 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 consists of a Daml driver that utilises and underlying Synchronization Technology to either implement the Daml ledger directly, or run the Canton protocol.

Canton Ledger

A Canton ledger is a privacy-enabled Daml ledger implemented using the Canton application, nodes, and protocol.

5.1.1.4 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.

5.1.1.5 Synchronization Technology

The syncronization 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.

5.1.1.6 Daml Drivers

Daml drivers enable a *ledger* to be implemented on top of different *synchronization technologies*; a database or distributed ledger technology.

5.1.2 Daml Language Concepts

5.1.2.1 Contract

A **contract** is an item 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 it is that the contract 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 stay 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.

5.1.2.2 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*.

5.1.2.3 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 you 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 choices 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.

5.1.2.4 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 *Contract keys*.

5.1.2.5 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 1: 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 is giving authorization to an update in one of two ways:

It is the signatory of the contract that contains the update action.

It is 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.

5.1.2.6 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*.

5.1.2.7 Agreement

An **agreement** is part of a *contract*. It is 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*.

5.1.2.8 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*.

5.1.2.9 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*.

5.1.2.10 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 *2 Testing templates using Daml Script*.

5.1.2.11 Contract key

A **contract key** allows you to uniquely identify a *contract* of a particular *template*, similarly 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 *Contract keys*.

5.1.2.12 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*.

5.1.3 Developer tools

5.1.3.1 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)*.

5.1.3.2 Studio

Daml Studio is a plugin for Visual Studio Code, and is the IDE for writing Daml code.

See *Daml Studio*.

5.1.3.3 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*.

5.1.3.4 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*.

5.1.4 Building applications

5.1.4.1 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 *Application architecture* page.

5.1.4.2 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 leger 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*.

5.1.4.3 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*.

5.1.4.4 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.

5.1.4.5 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*.

5.1.4.6 Participant Node

The participant node is a server that provides users a consistent programmatic access to a ledger through the *Ledger API*. The participant nodes handles 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.

5.1.4.7 Sub-Transaction Privacy

Sub-transaction privacy is where participants to 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.

5.1.4.8 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)

5.1.4.9 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.

5.1.4.10 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.

5.1.5 Canton Concepts

5.1.5.1 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*.

5.1.5.2 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.

5.1.5.3 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.

5.1.5.4 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 aggregates them into a single result. The mediator does not learn about the content of the transaction, they only learn about the involved participants.

5.1.5.5 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.

5.1.5.6 Domain Identity Manager

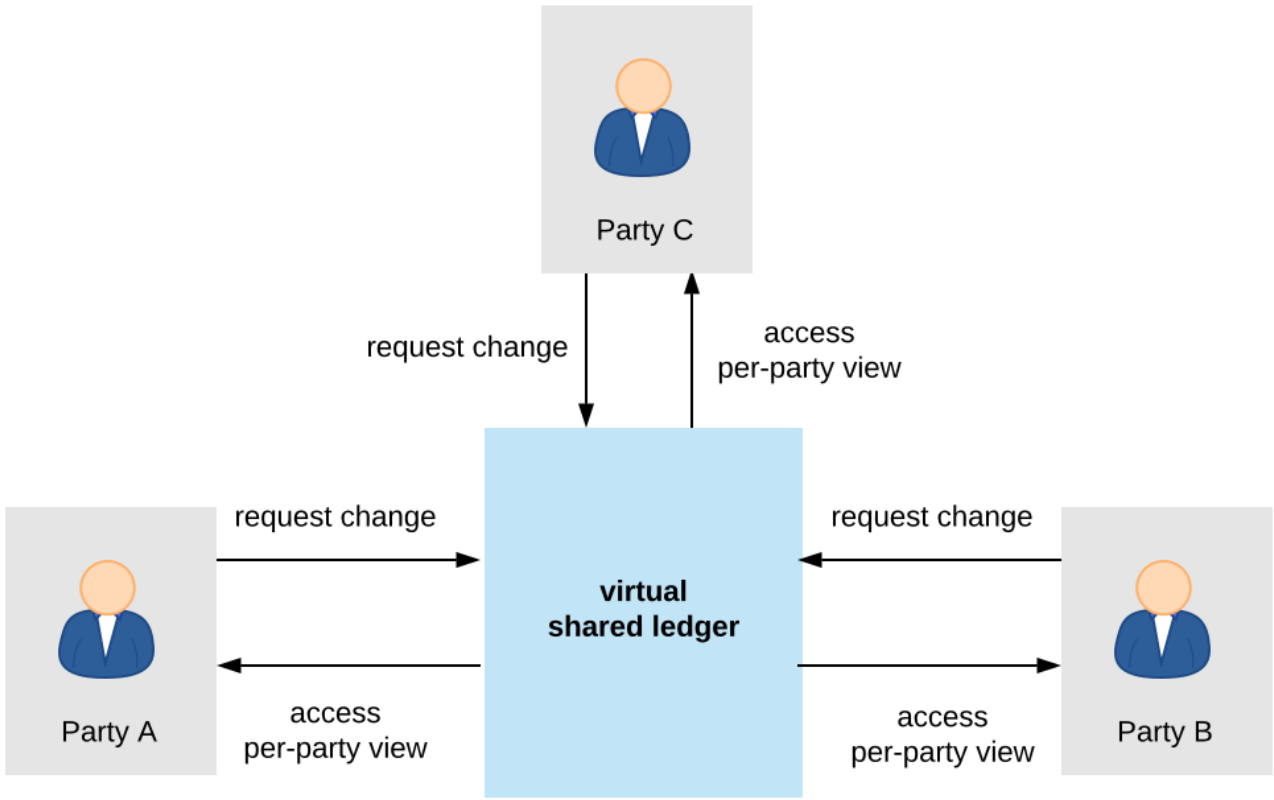
The Domain Identity Manager is a service provided by the *domain*, used by the *Canton protocol*. Participants join a new domain by registering with the domain identity manager. The domain identity manager establishes a consistent identity state among all participants. The domain identity manager only forwards identity updates. It can not invent them.

5.1.5.7 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.

5.2 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.

5.2.1 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*.

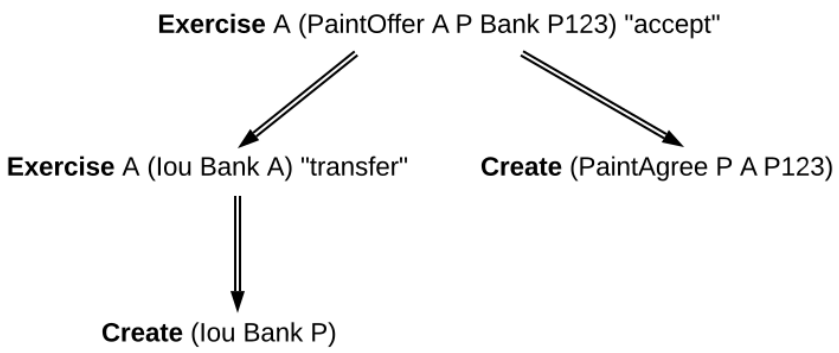
5.2.1.1 Actions and Transactions

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.

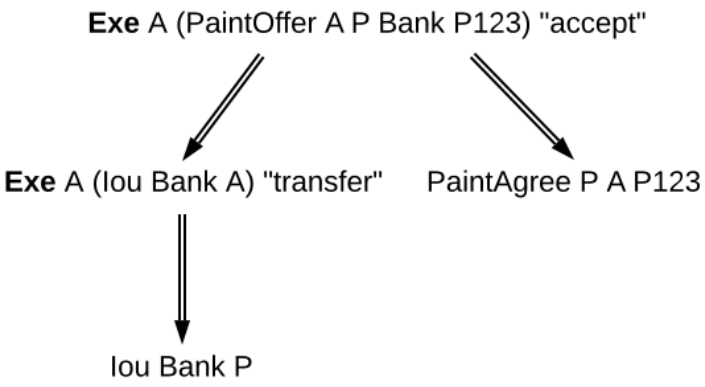
The following EBNF-like grammar summarizes the structure of actions and transactions. Here, s | t represents the choice between *s* and *t*, s t represents *s* followed by *t*, and s* represents the repetition of *s* zero or more times. The terminal 'contract' denotes the underlying type of contracts, 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'
```

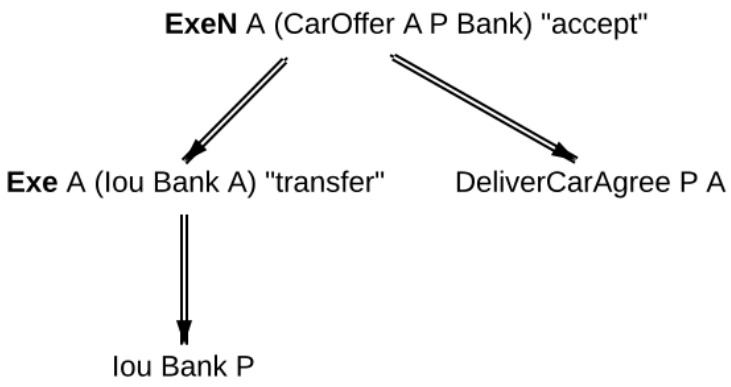
The visual notation presented earlier captures actions precisely with conventions that:

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.

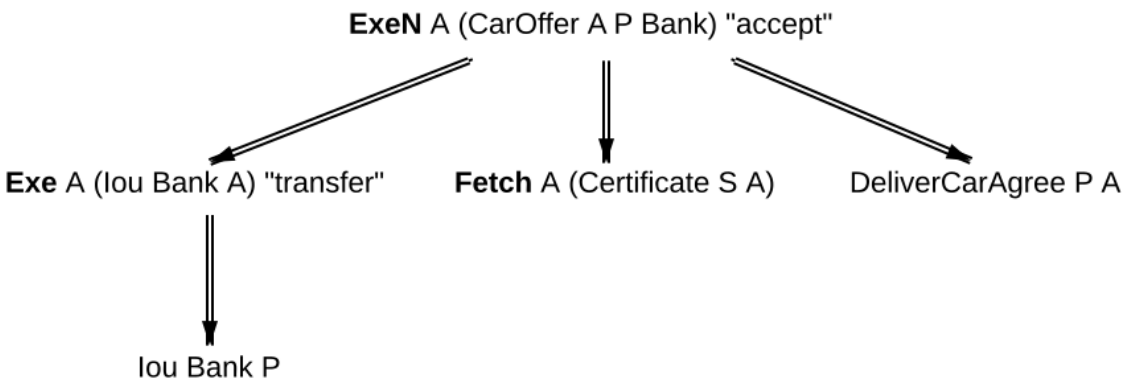
An alternative shorthand notation, shown below uses the abbreviations **Exe** and **ExeN** for exercises, and omits the **Create** labels on create actions.



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:

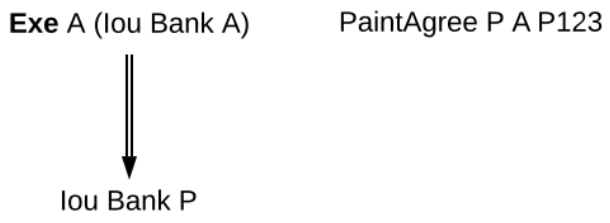
PaintOffer houseOwner painter obligor refNo Intuitively an offer (with a reference number) by which the painter proposes to the house owner to paint her house, in exchange for a single IOU token issued by the specified obligor.

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).

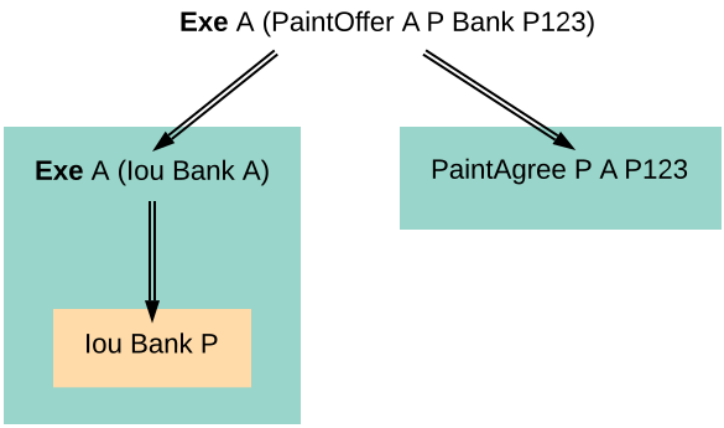
In practice, multiple IOU contracts can exist between the same *obligor* and *owner*, in which case each contract should have a unique identifier. However, in this section, each contract only appears once, allowing us to drop the notion of identifiers for simplicity reasons.

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.

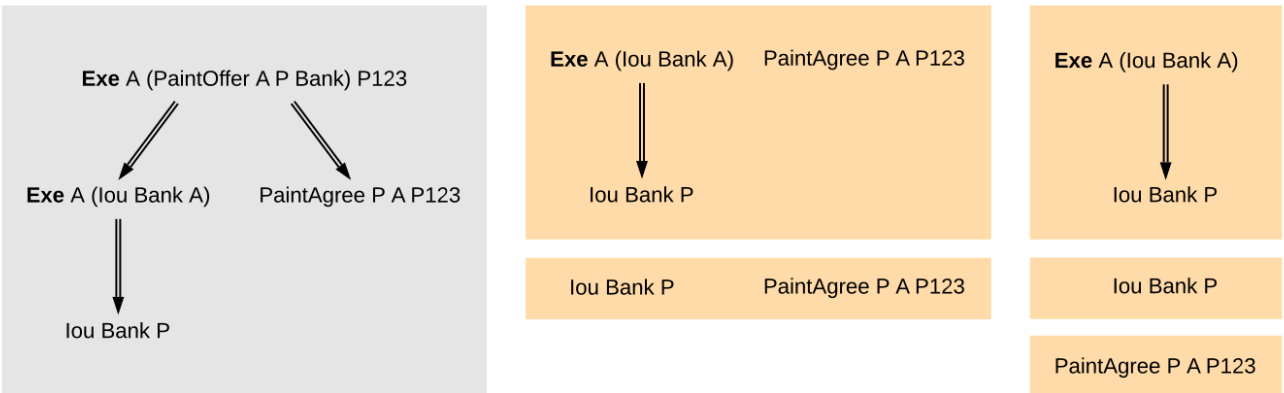


For an action *act*, its **proper subactions** are all actions in the consequences of *act*, together with all of their proper subactions. Additionally, *act* is a (non-proper) **subaction** of itself.

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.

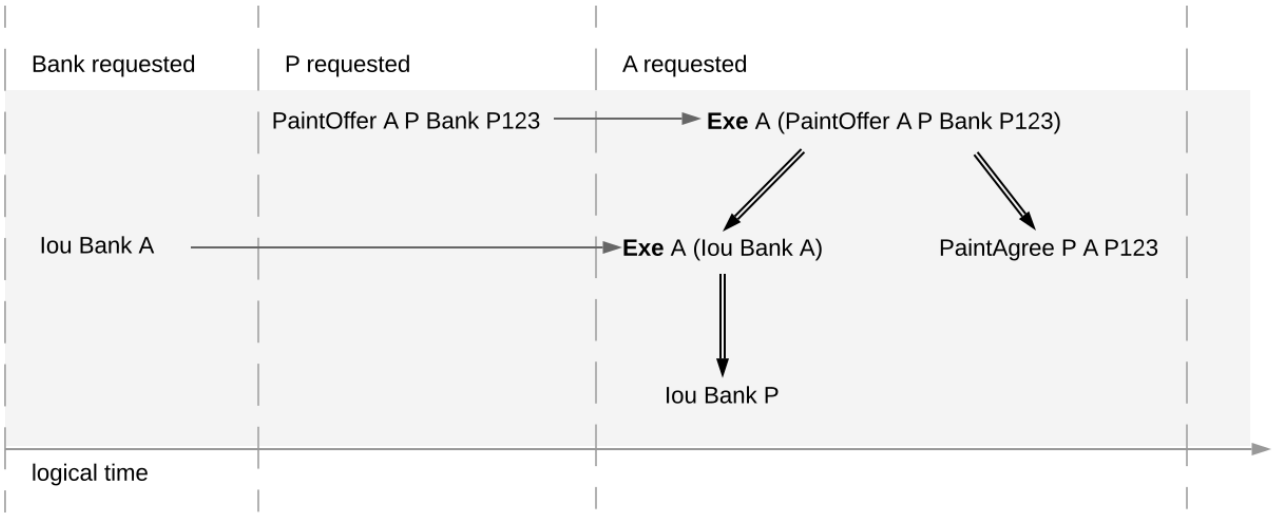
5.2.1.2 Ledgers

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.

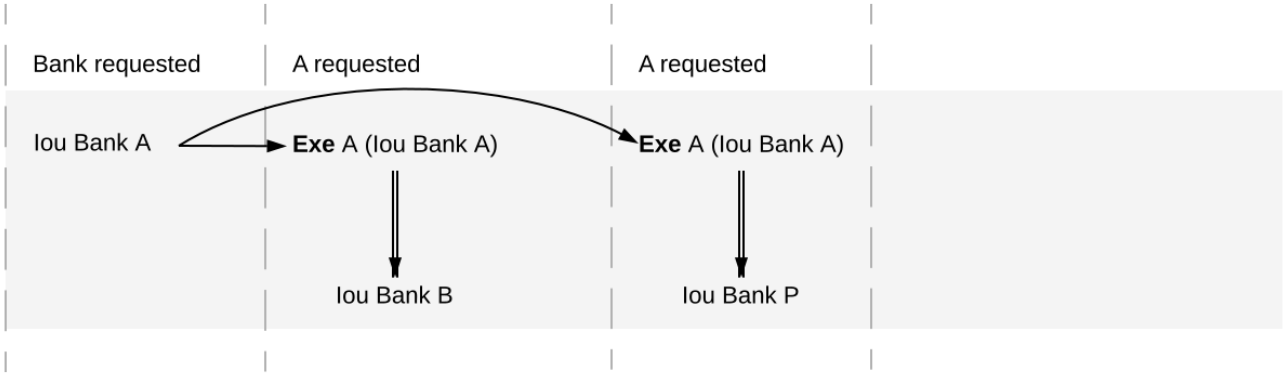


Fig. 1: Alice spending her IOU twice (double spend), once transferring it to *B* and once to *P*.

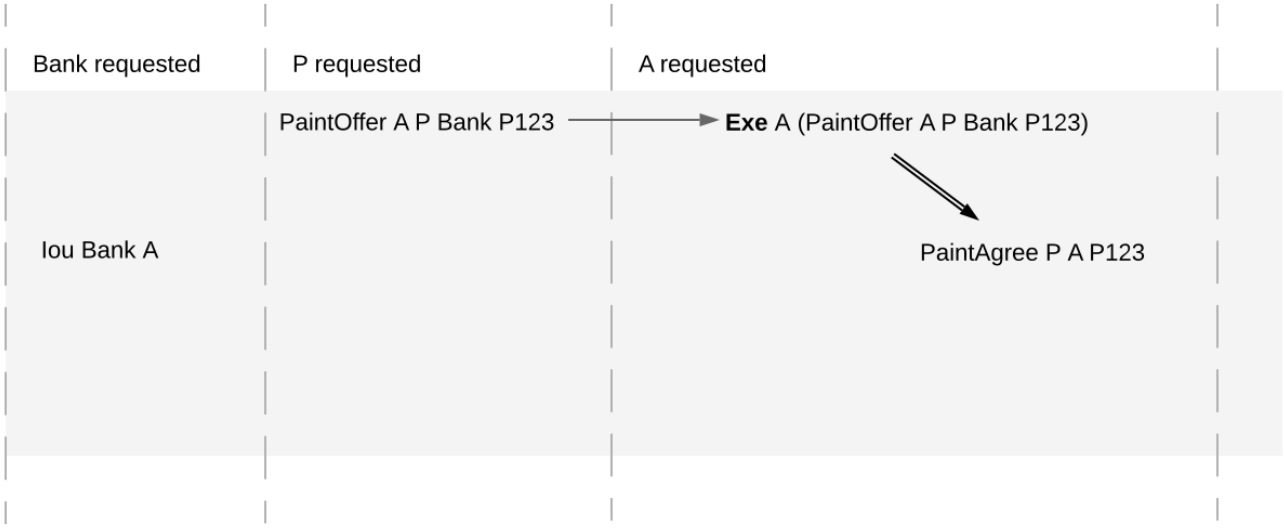


Fig. 2: Alice changing the offer's outcome by removing the transfer of the *Iou*.

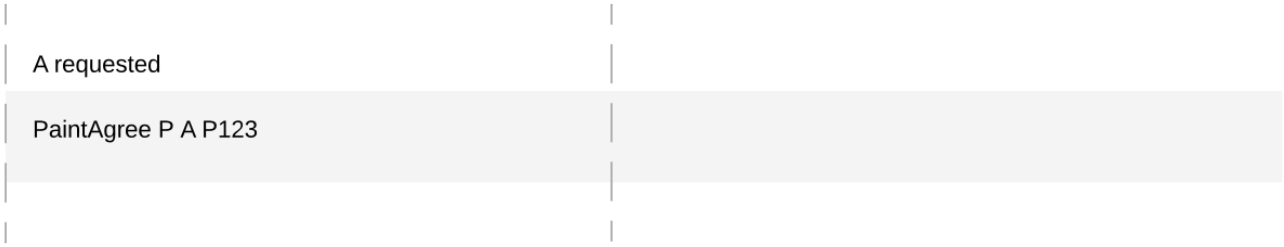


Fig. 3: An obligation imposed on the painter without his consent.

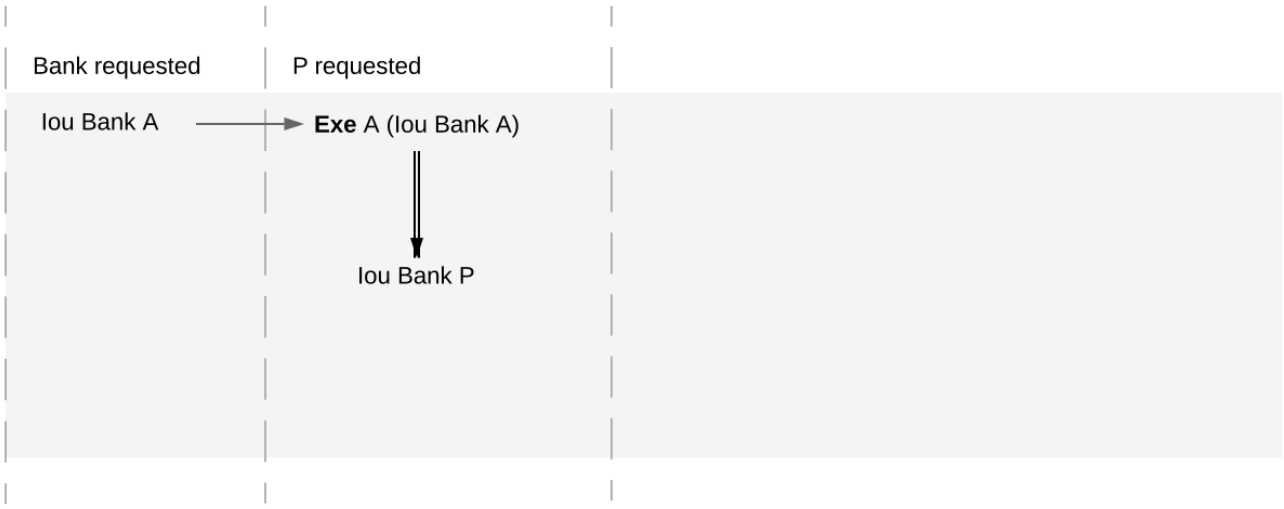


Fig. 4: Painter stealing Alice's IOU. Note that the ledger would be intuitively permissible if it was Alice performing the last commit.

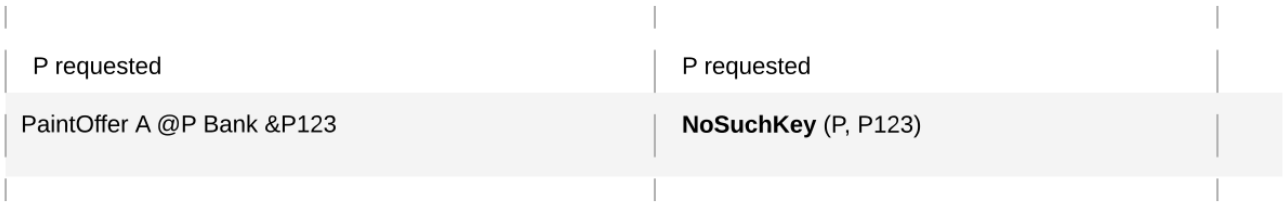


Fig. 5: Painter falsely claiming that there is no offer.

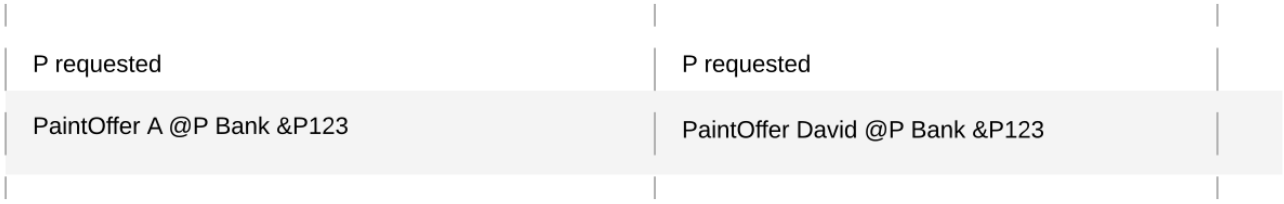


Fig. 6: Painter trying to create two different paint offers with the same reference number.

5.2.2 Integrity

This section addresses the question of who can request which changes.

5.2.2.1 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.

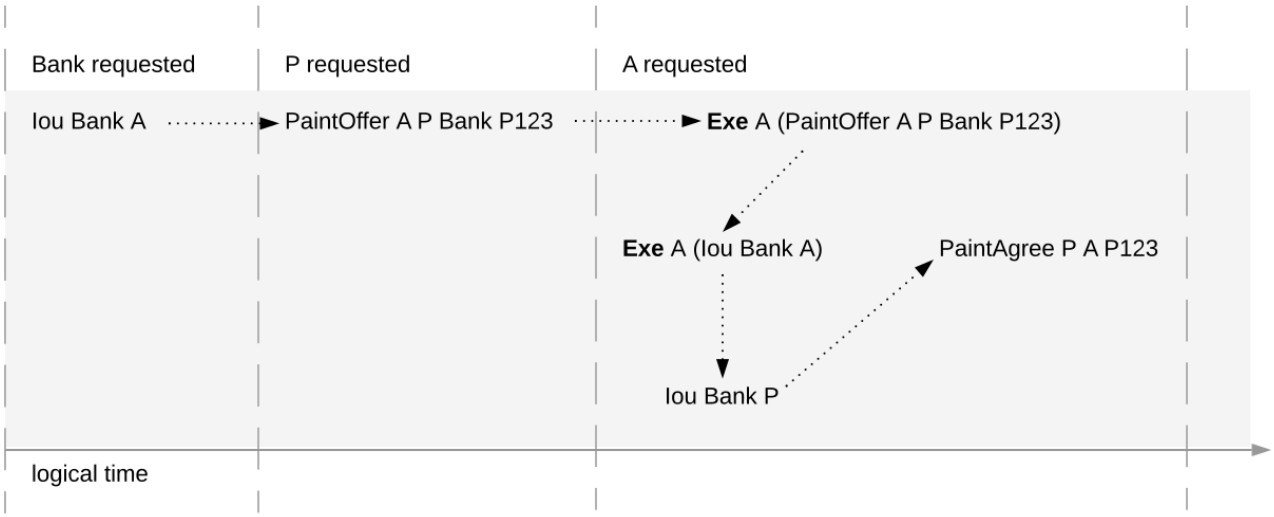
Only the last of these conditions depends on the party (or parties) requesting the change; the other two are general.

5.2.2.2 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*.

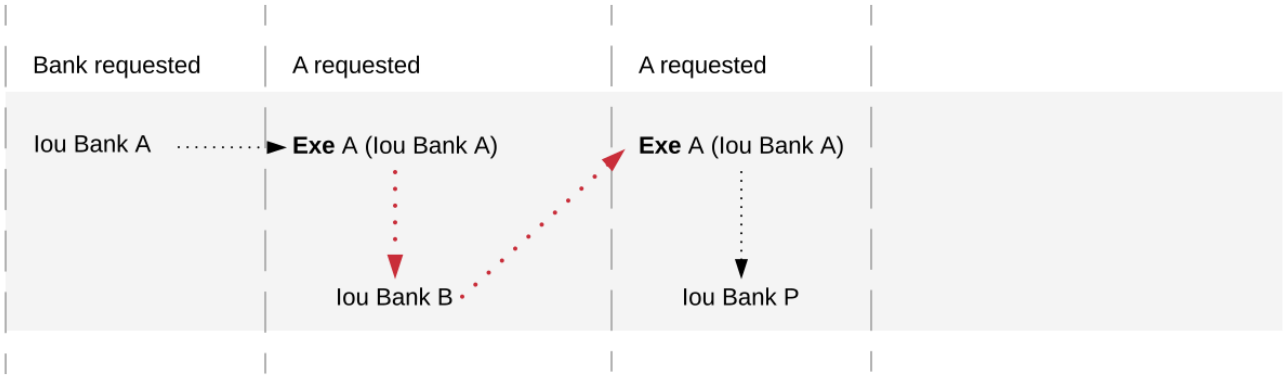
Contract consistency

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*.

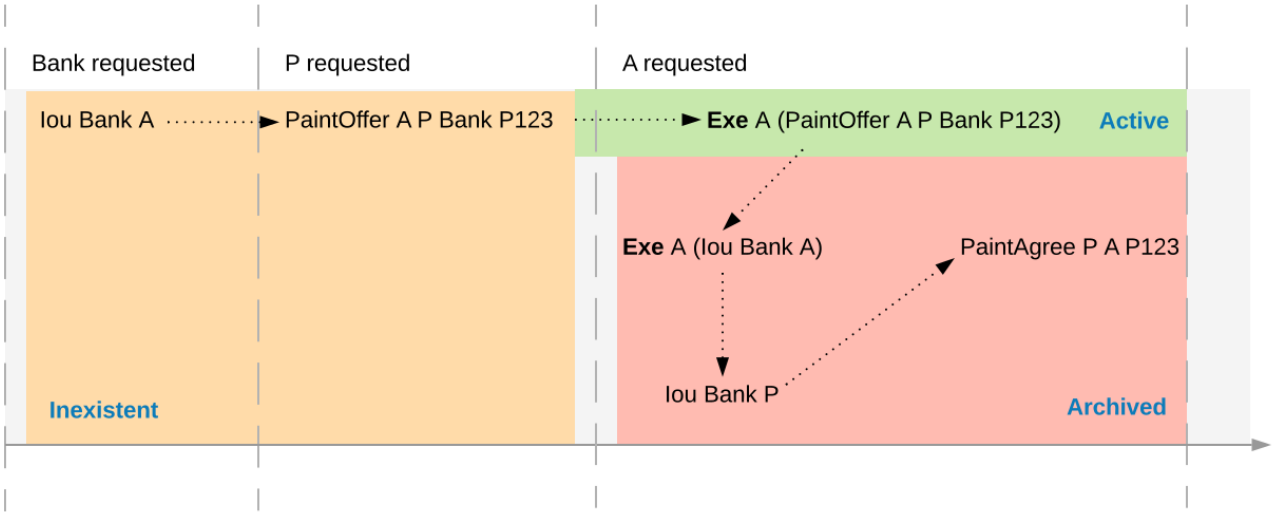


Fig. 7: Activeness of the *PaintOffer* contract

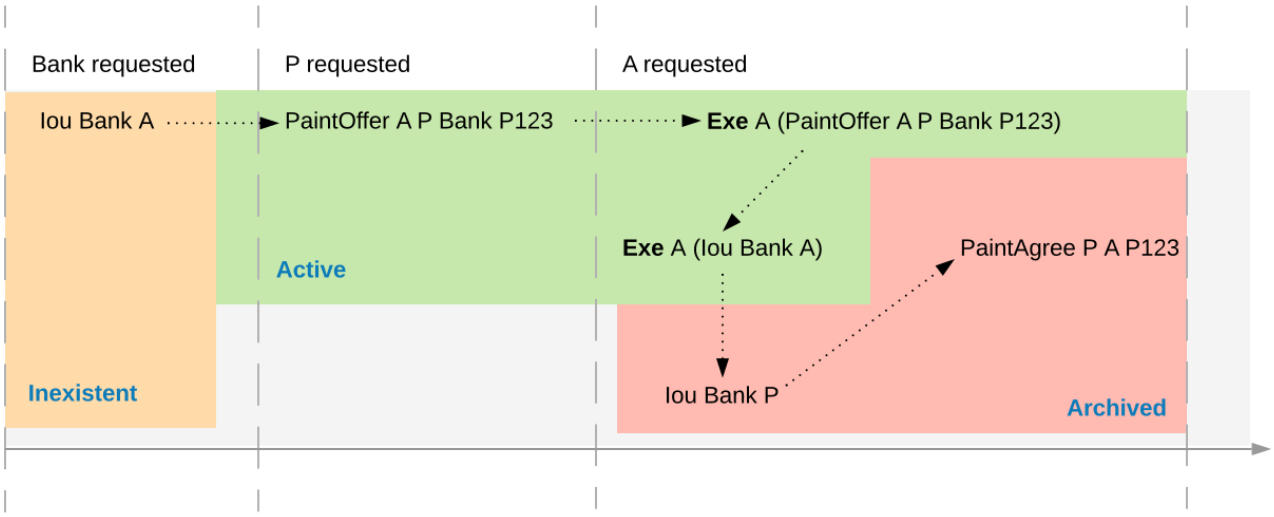


Fig. 8: Activeness of the *Iou Bank A* contract

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

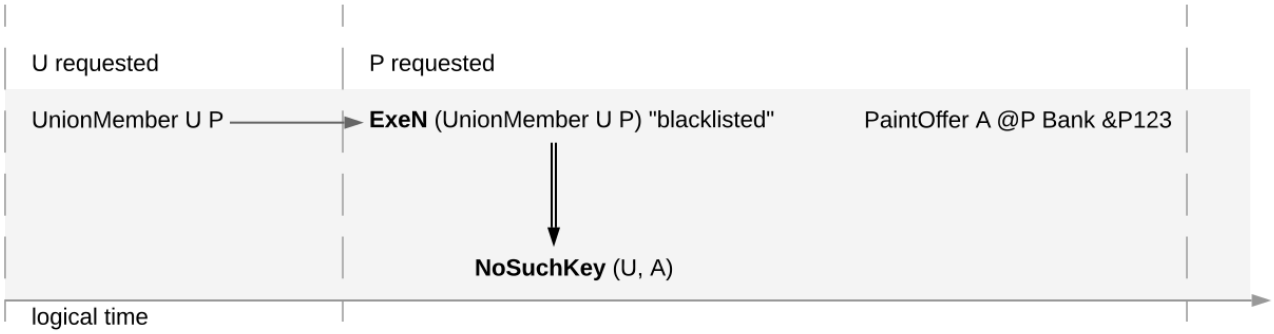
> If *act* is a **Create** action or **NoSuchKey** assertion, then *s* is **free** or **unknown**.
> If *act* is an **Exercise** or **Fetch** action on some contract *c*, then *s* is **assigned** to *c* or **unknown**.

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.

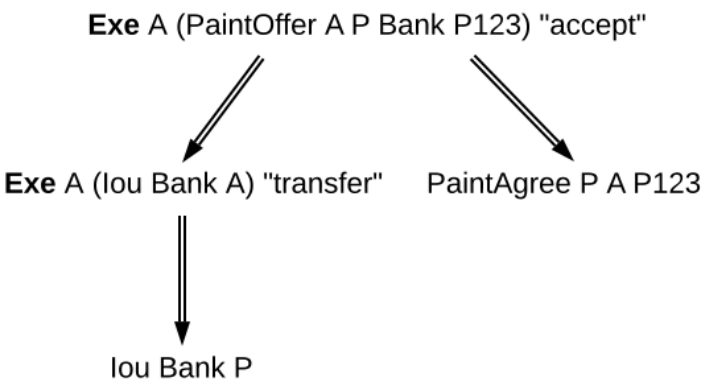


Ledger consistency

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.

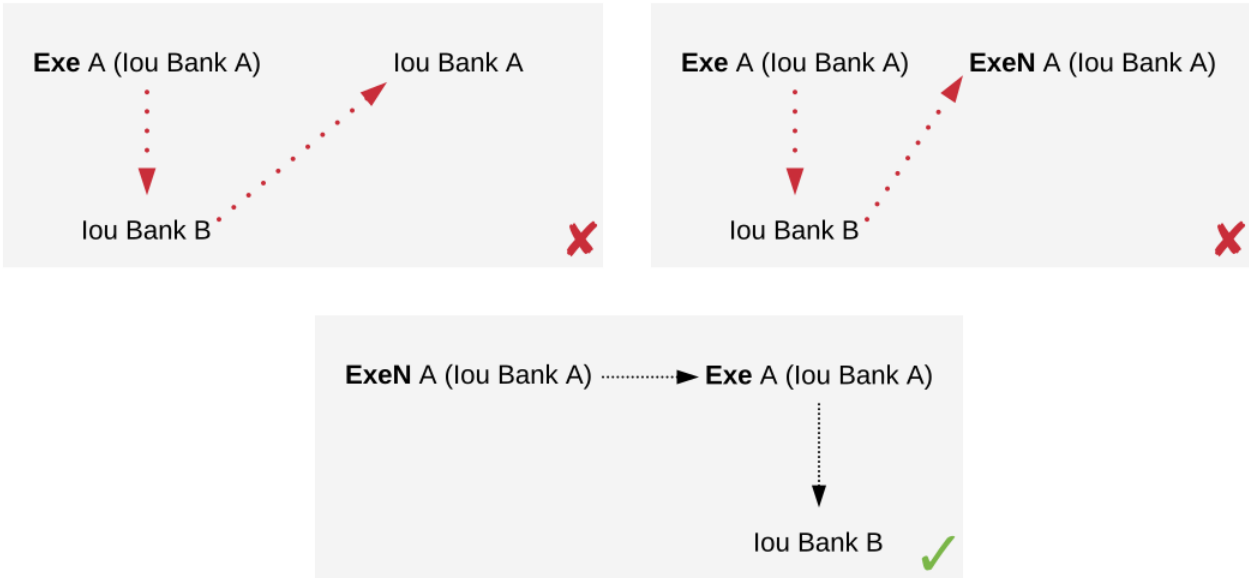


Fig. 9: The first two transactions violate the conditions of internal consistency. The first transaction creates the *Iou* after exercising it consumingly, violating both conditions. The second transaction contains a (non-consuming) exercise on the *Iou* after a consuming one, violating the second condition. 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.

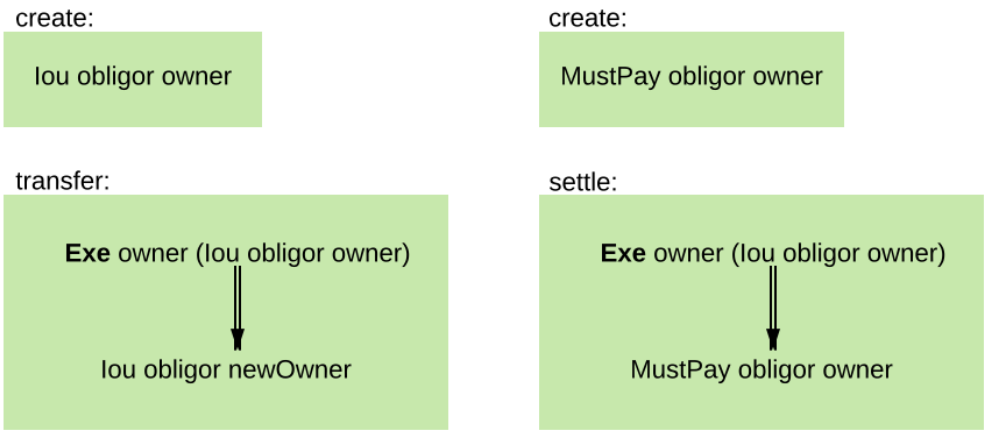
Definition input key A key *k* is an **input key** to an internally consistent transaction if the first action *act* on *k* is either a **Create** action or a **NoSuchKey** assertion.

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.

5.2.2.3 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.

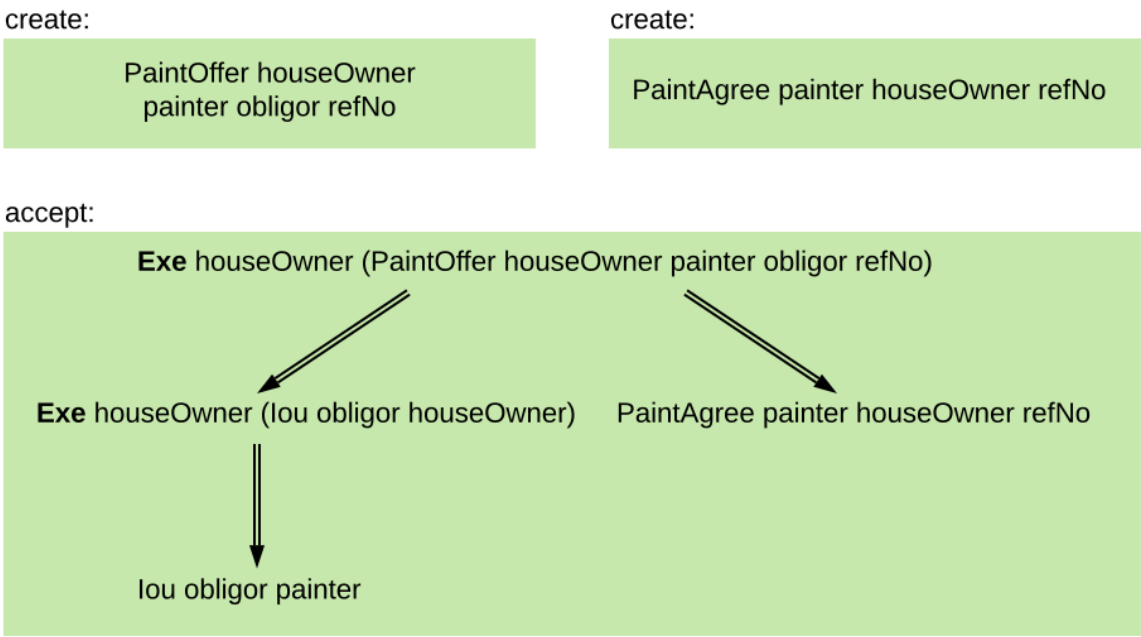
For example, the set of allowed actions on IOU contracts could be described as follows.



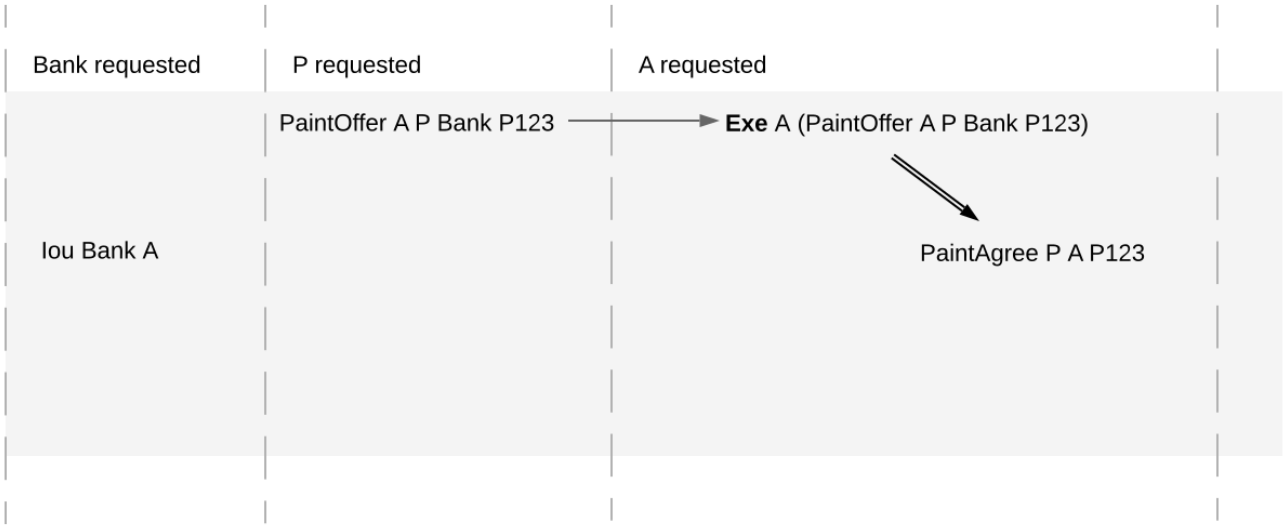
The boxes in the image are templates in the sense that the contract parameters in a box (such as obligor or owner) can be instantiated by arbitrary values of the appropriate type. To facilitate understanding, each box includes a label describing the intuitive purpose of the corresponding set of actions. 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.



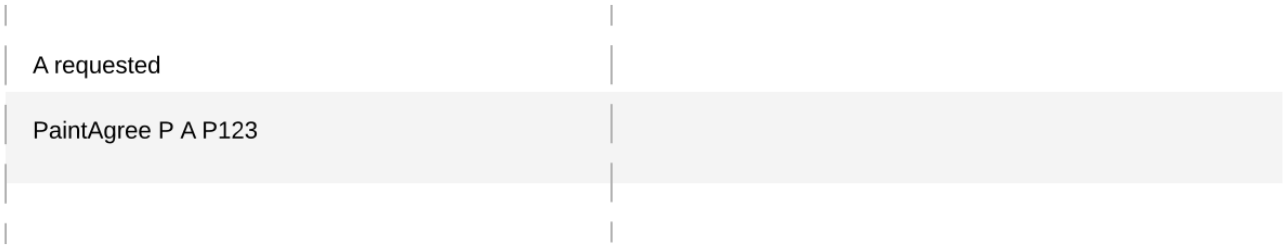
The problem with example where Alice changes the offer's outcome to avoid transferring the money now becomes apparent.



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.

5.2.2.4 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

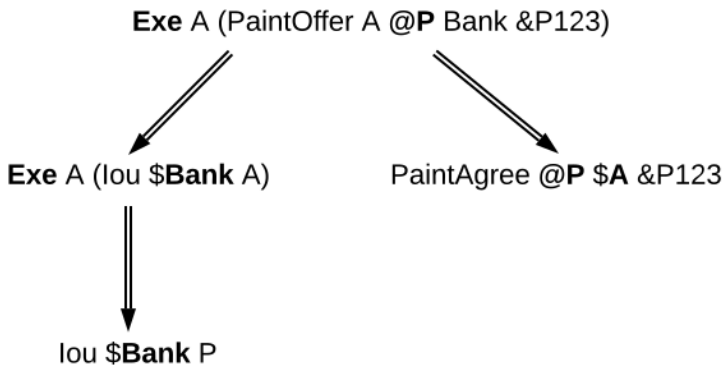
To capture these elements of real-world contracts, the **contract model** additionally specifies, for each contract in the system:

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:

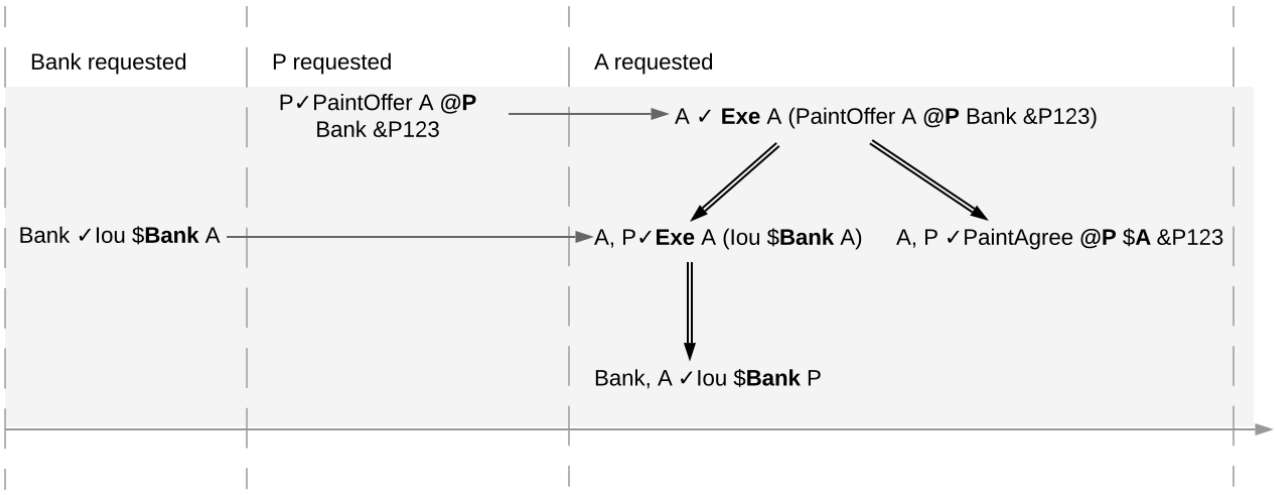
1. the required authorizers of a **Create** action on a contract *c* are the signatories of *c*.
2. the required authorizers of an **Exercise** or a **Fetch** action are its actors.
3. the required authorizers of a **NoSuchKey** assertion are the maintainers of the key.

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.

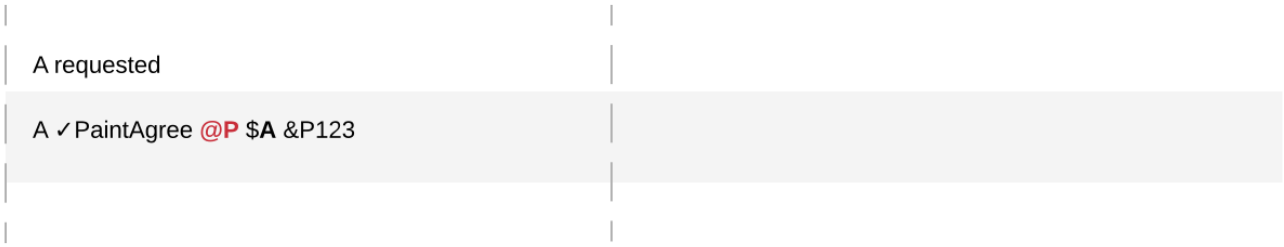
In the visualizations below, ✓ *act* denotes that the parties authorize the action *act*. The resulting authorizations are shown below.



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.

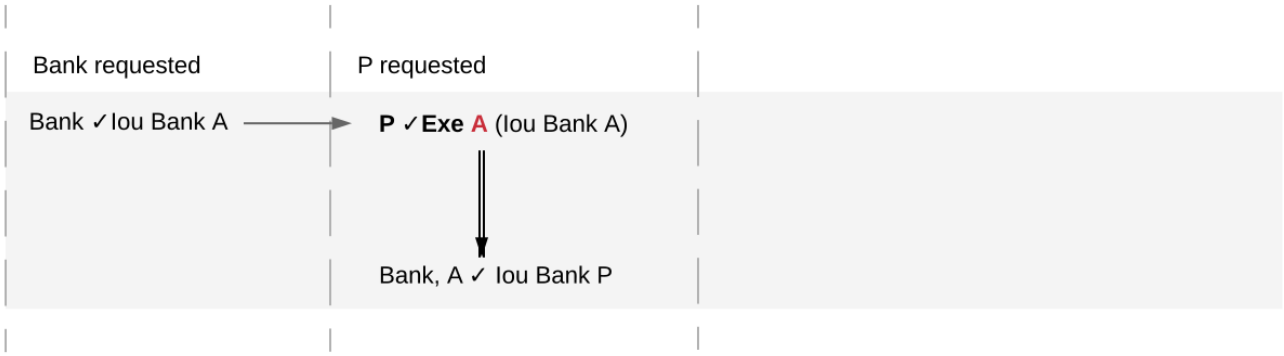
The third commit is more complicated. First, Alice authorizes the exercise on the paint offer by requesting it. She is the only actor on this exercise, so this complies with the authorization requirement. Since the painter is the signatory of the paint offer, and Alice the actor of the exercise, they 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 paint agreement, which has Alice and the painter as signatories. Since they both authorize this action, this is also permissible. Finally, the creation of the new IOU (for P) is a consequence of the exercise on the old one (for A). As the old IOU was signed by the bank, and as Alice was the actor of the exercise, the bank and Alice jointly authorize the creation of the new IOU. Since the bank is the sole signatory of this IOU, this action is also permissible. Thus, the entire third commit is also well-authorized, and then 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*.

5.2.2.5 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:

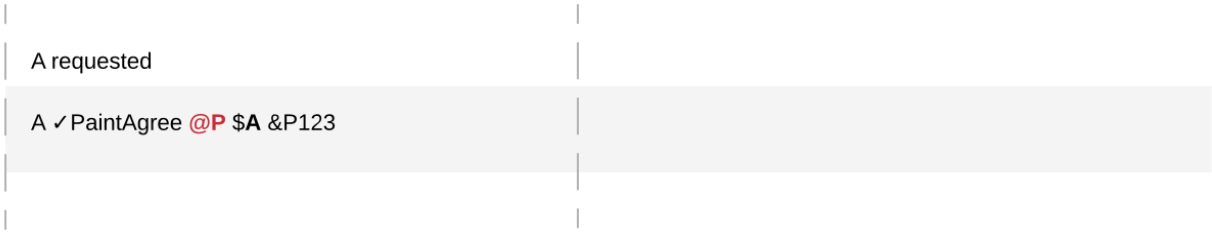
> a real-world offer, whose consequences (both on- and off-ledger) are specified by the **Exercise** actions on the contract allowed by the contract model, and
> a real-world contract proper , specified through the contract's (optional) agreement text.

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

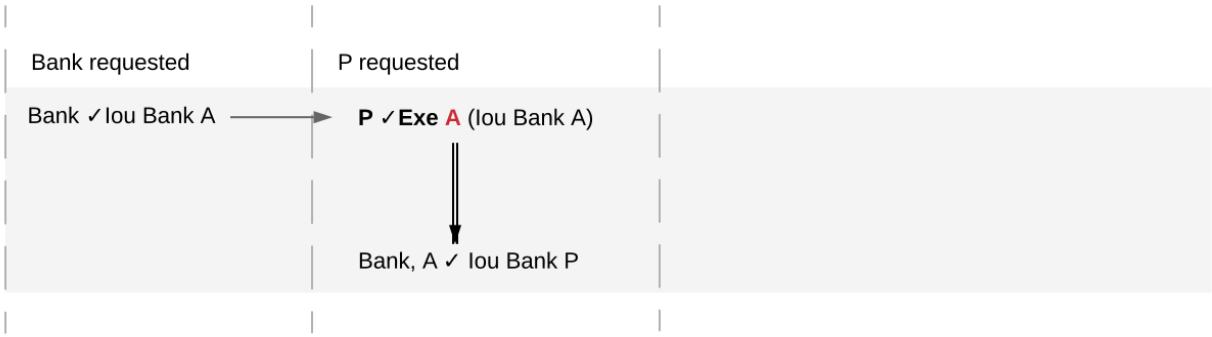
Daml ledgers, a party's rights according to a contract model are the exercise actions the party can perform according to the authorization and conformance rules.

Finally, validity conditions ensure three important properties of the Daml ledger model, that mimic the contract law.

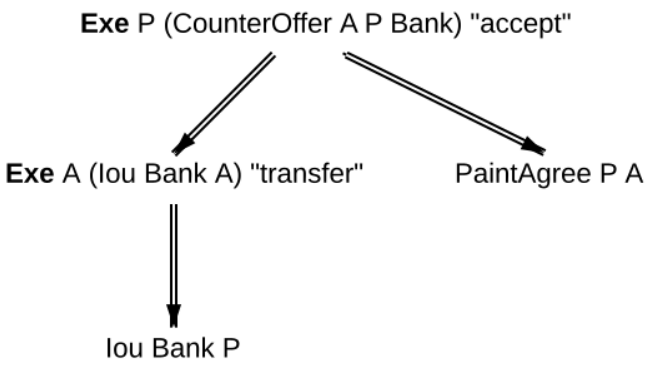
1. **Obligations need consent**. Daml ledgers follow the offer-acceptance pattern of the contract law, and thus ensures that all ledger contracts are formed voluntarily. For example, the following ledger is not valid.



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:

. . .	Bank commits
. . .	**Exe** Bank (MustPay Bank A)
MustPay Bank A	

That is, this ledger is invalid, as the action above is not conformant to the contract model.

5.2.3 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.

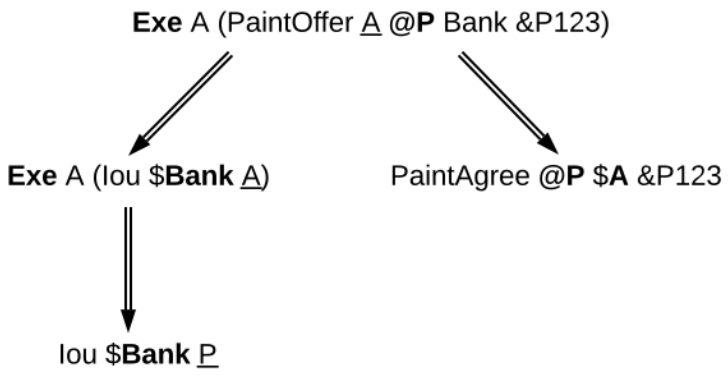
To make this more precise, a stakeholder concept is needed.

5.2.3.1 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.



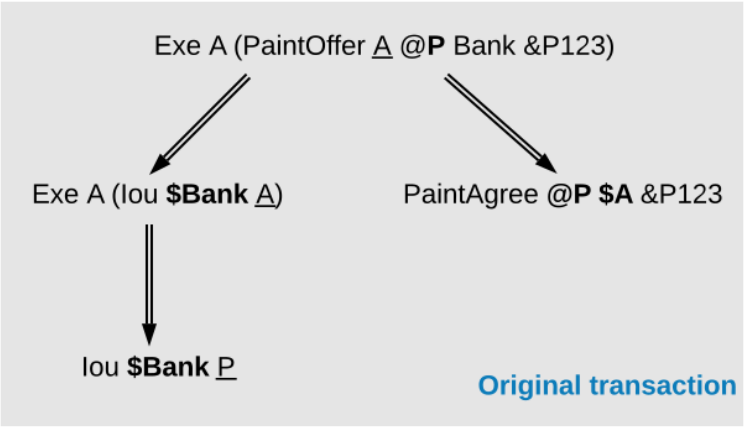
5.2.3.2 Choice Observers

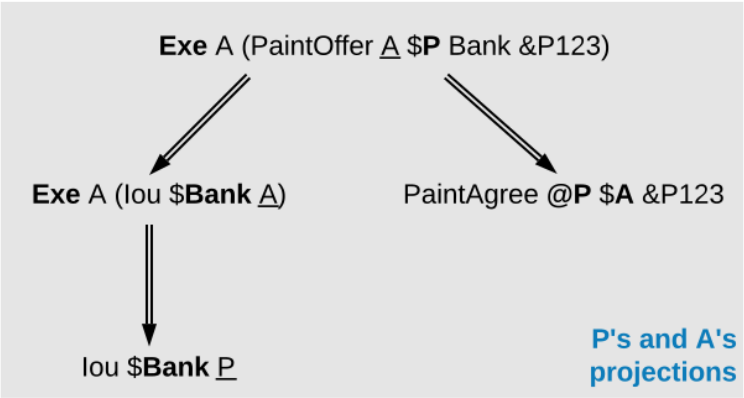
In addition to contract observers, the contract model can also specify **choice observers** on individual **Exercise** actions. Choice observers get to see a specific exercise on a contract, and to view its consequences. Choice observers are not considered stakeholders of the contract, they only affect the set of informees on an action, for the purposes of projection (see below).

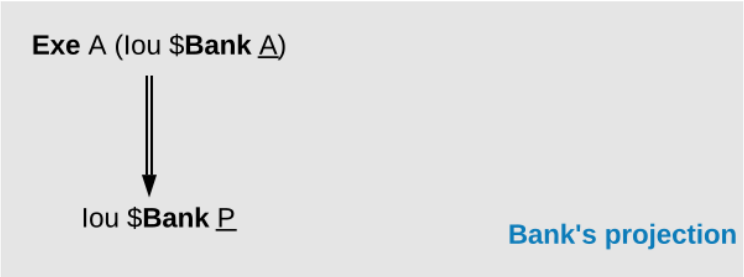
5.2.3.3 Projections

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.

5.2.3.4 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.

5.2.3.5 Divulgence: When Non-Stakeholders See Contracts

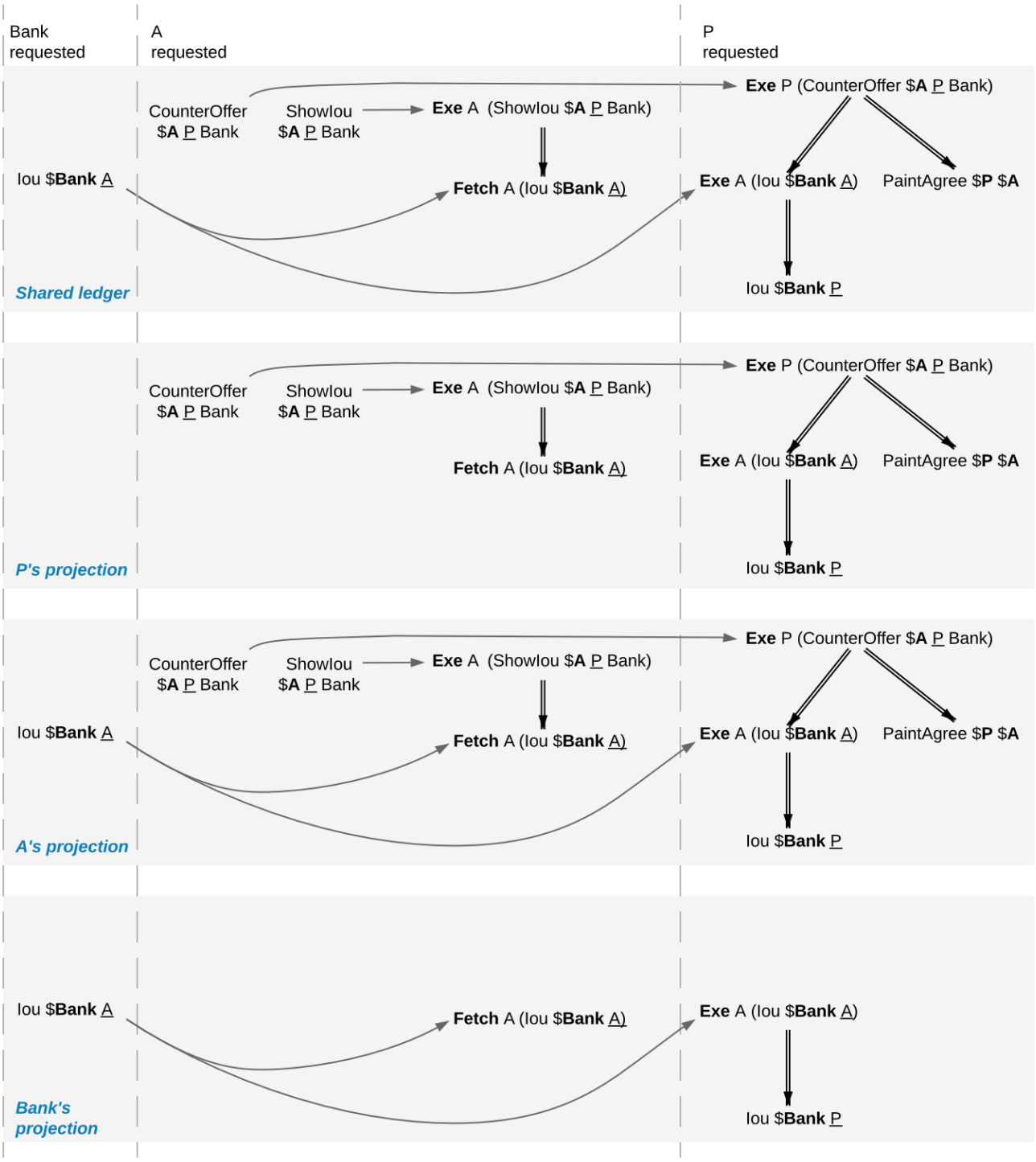
The guiding principle for the privacy model of Daml ledgers is that contracts should only be shown to their stakeholders. However, ledger projections can cause contracts to become visible to other parties as well.

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.

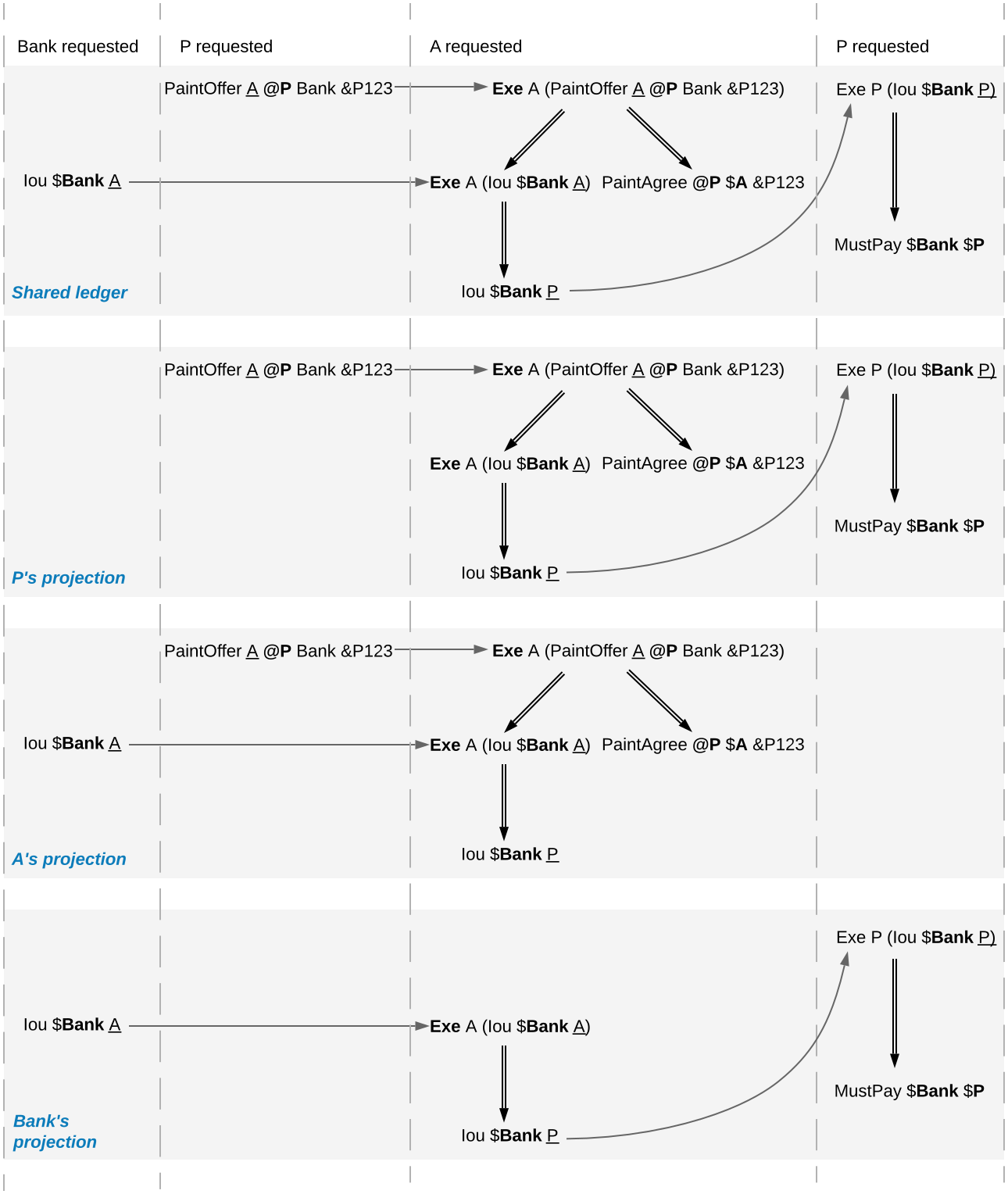
Showing contracts to non-stakeholders through ledger projections is called **divulgence**. Divulgence is a deliberate choice in the design of Daml ledgers. In the paint offer example, the only proper way to accept the offer is to transfer the money from Alice to the painter. Conceptually, at the instant 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.

5.2.4 Daml: Defining 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.

5.2.5 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.

5.2.5.1 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.

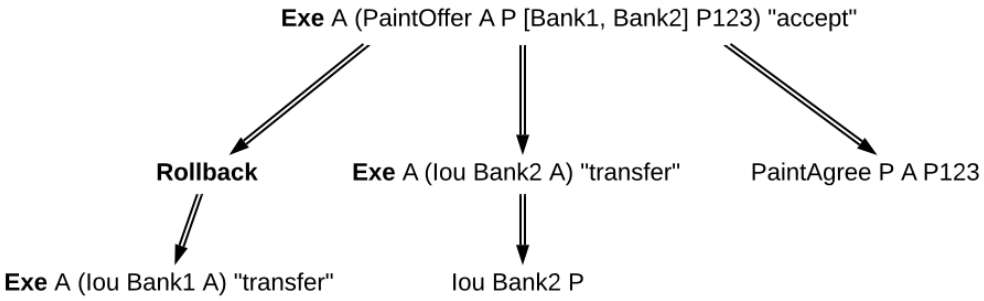
To support this in our ledger model, we need to modify the transaction structure to indicate which subtransactions were rolled back. We do this by introducing **rollback nodes** in the transaction. Each rollback node contains a rolled back subtransaction. Rollback nodes are not considered ledger actions.

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'
```

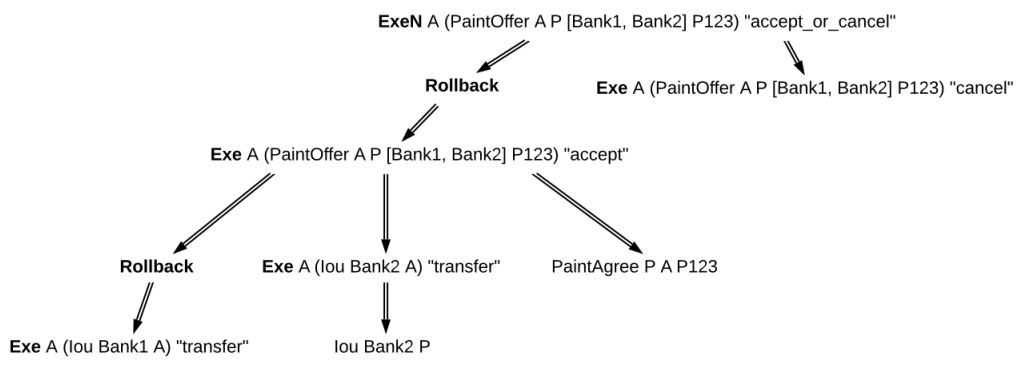
Note that *Action* and *Kind* have the same definition as before, but since *Transaction* may now contain rollback nodes, this means that an *Exercise* action may have a rollback node as one of its consequences.

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.



5.2.5.2 Consistency

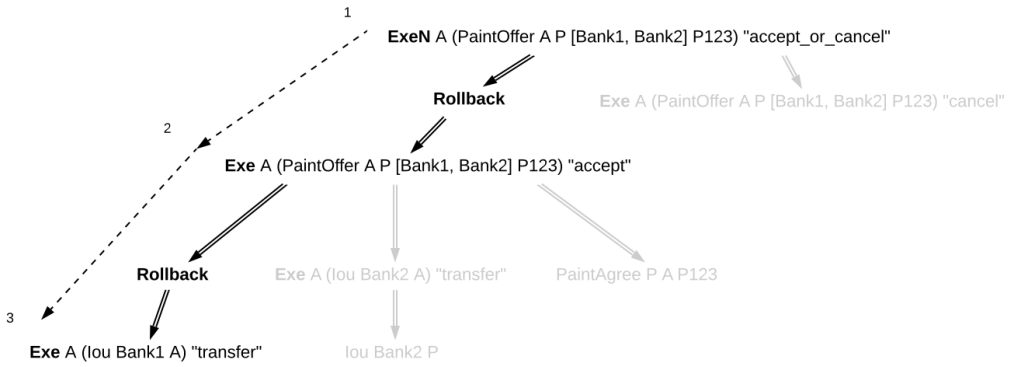
In the previous section on *consistency*, we defined a before-after relation on ledger actions. This notion needs to be revised in the presence of rollback nodes. It is no longer enough to perform a preorder traversal of the transaction tree, because the actions under a rollback node cannot affect actions 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.

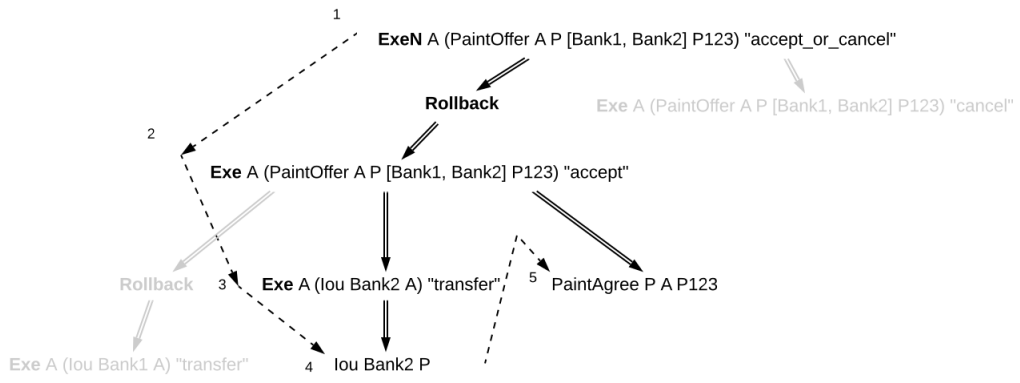
So, we now define the before-after relation as a partial order, rather than a total order, on all the actions of a transaction. This relation is defined as follows: *act1* comes before *act2* (equivalently, *act2* comes after *act1*) if and only if *act1* appears before *act2* in a preorder traversal 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*.

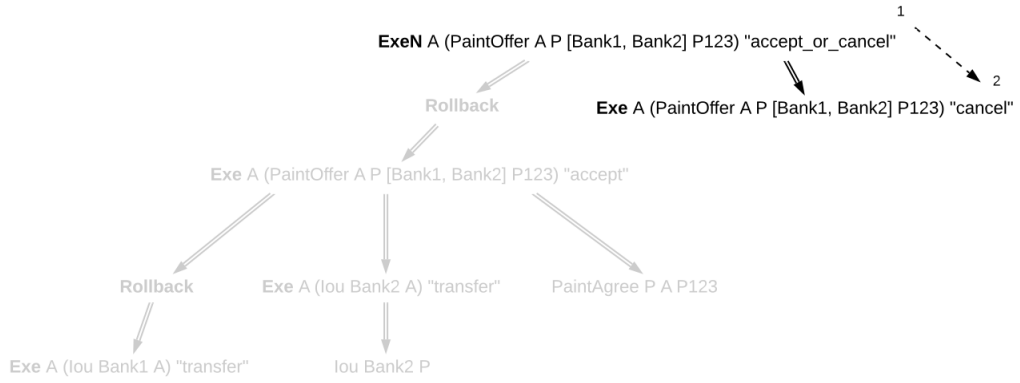
In the example above, neither consuming exercise comes after the other. They are part of separate continuities, so they don't introduce inconsistency. Here are three continuities implied by the before-after relation. The first:



The second:



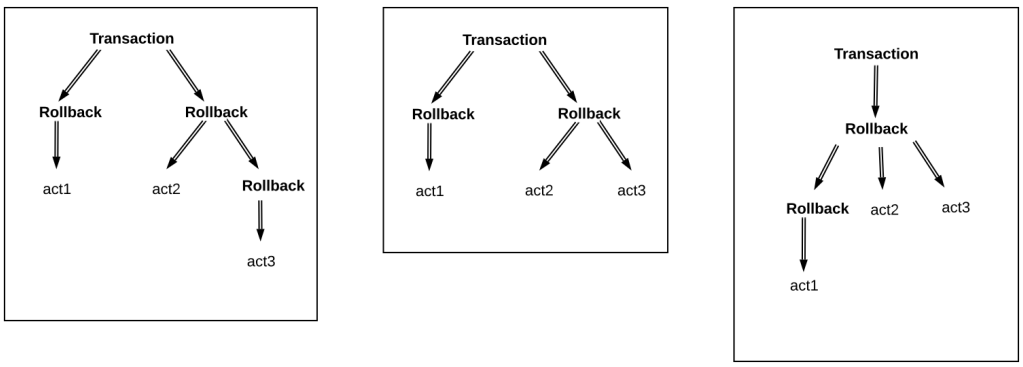
And the third:



As you can see, in each of these continuities, no contract was consumed twice.

5.2.5.3 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*):



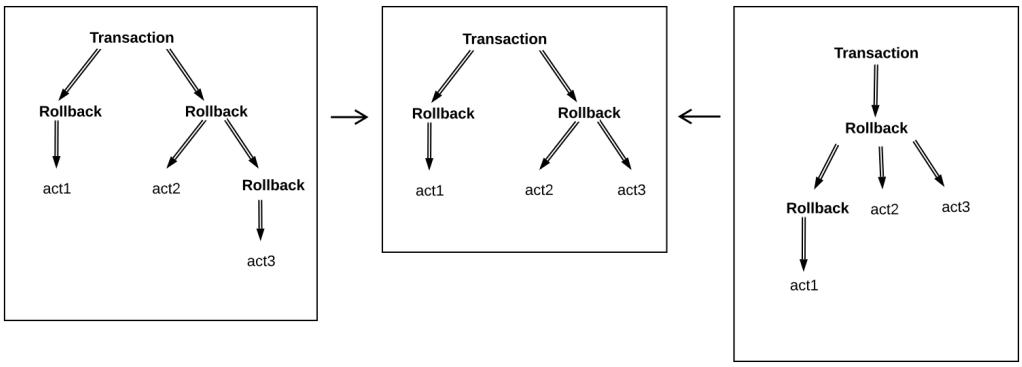
Because of this, these three transactions are equivalent. More generally, two transactions are equivalent if:

> 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.

5.2.5.4 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.

5.2.5.5 Privacy

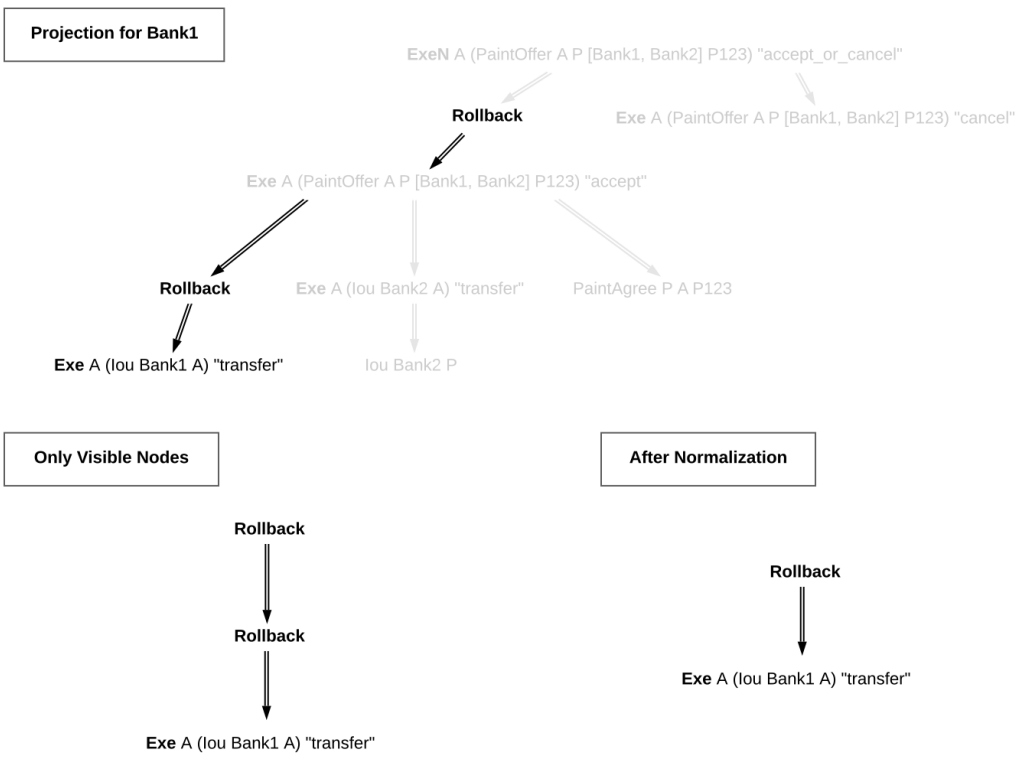
Rollback nodes also have an interesting effect on the notion of privacy in the ledger model. When projecting a transaction for a party *p*, it's necessary to preserve some of the rollback structure of the transaction, even if *p* does not have the right to observe every action under it. For example, 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.

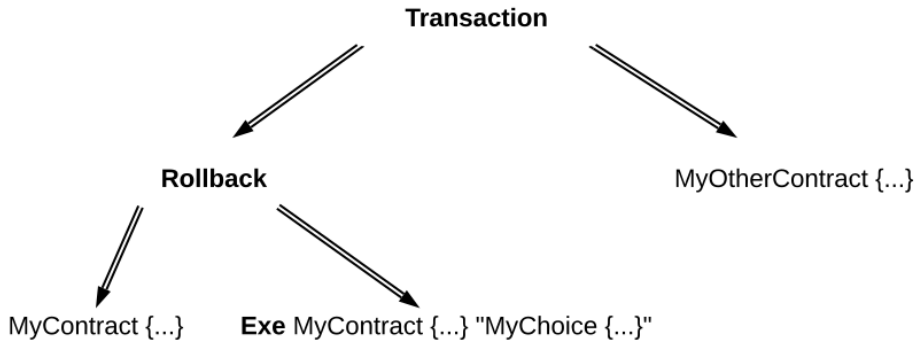
5.2.5.6 Relation to Daml Exceptions

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.

5.3 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.

5.3.1 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.

5.3.1.1 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.

5.3.1.2 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*.

5.3.1.3 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.

5.3.2 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.

5.3.2.1 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.

5.3.2.2 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.

5.3.2.3 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`.

5.3.2.4 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.

5.4 Time

The Daml language contains a function *getTime* which returns the current time. However, the notion of time comes with a lot of problems 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`.

5.4.1 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.

5.4.2 Record time

The *record time* `rt_TX` is another property of a transaction. It is timestamp with microsecond resolution, and is assigned by the ledger when the transaction is recorded on the ledger.

The record time should be an intuitive representation of real time, but the Daml ledger model does not prescribe how exactly the record time is assigned. Each ledger implementation might use a different way of representing time in a distributed setting - for details, contact your ledger operator.

5.4.3 Guarantees

The ledger time of 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.

5.4.4 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.

5.4.5 Assigning 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 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 record 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.

5.5 Causality and Local Ledgers

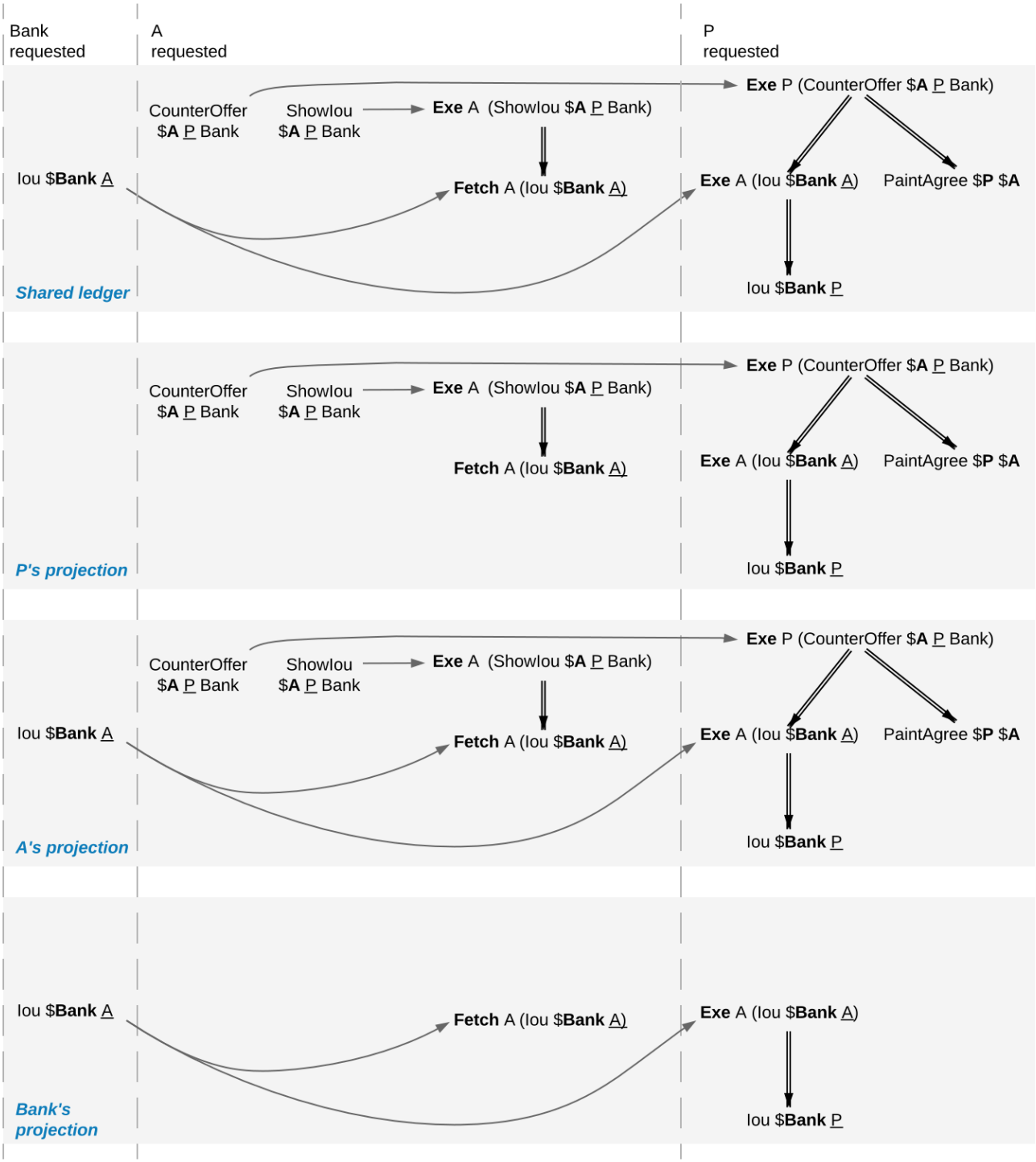
Daml ledgers do not totally order all transactions. So different parties may observe two transactions on different Participant Nodes in different orders via the *Ledger API*. Moreover, different Participant Nodes may output two transactions for the same party in different orders. This document explains 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*

5.5.1 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.



5.5.1.1 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.

5.5.1.2 Signatories of a contract and stakeholder actors see usages after the creation and before the archival.

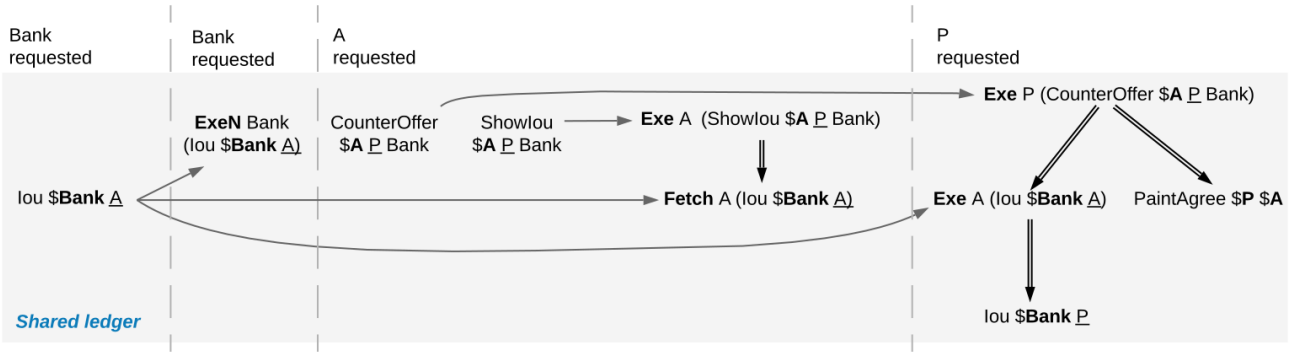
The *Fetch A (Iou Bank A)* action comes after the creation of the *Iou Bank A* and before its archival, for both Alice and the Bank, because the Bank is a signatory of the *Iou Bank A* contract and Alice is a stakeholder of the *Iou Bank A* contract and an actor on the **Fetch** action.

5.5.1.3 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*.

5.5.1.4 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.

5.5.1.5 Out-of-band causality is not respected.

The following examples assume that Alice splits up her commit into two as follows:

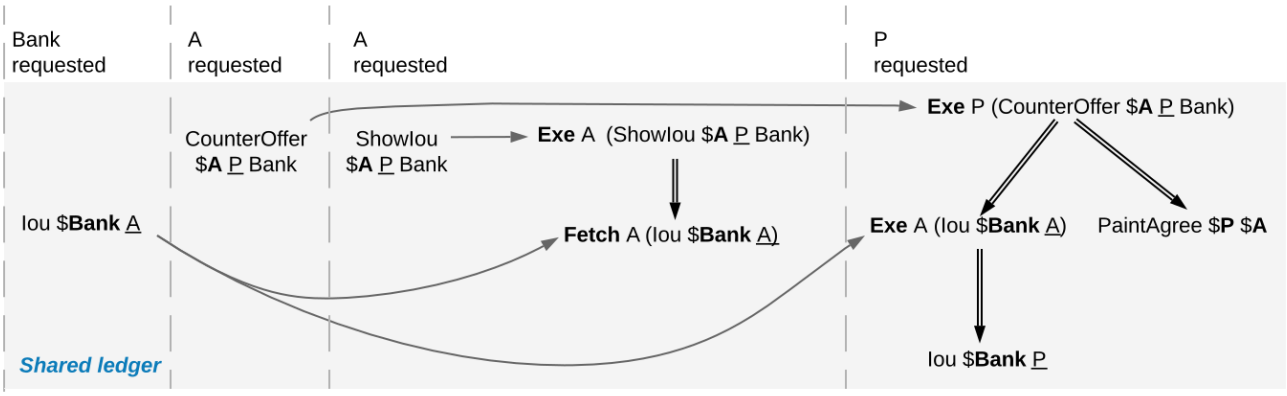


Fig. 10: Counteroffer workflow with four commits.

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.

5.5.1.6 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.

5.5.1.7 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.

5.5.2 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.

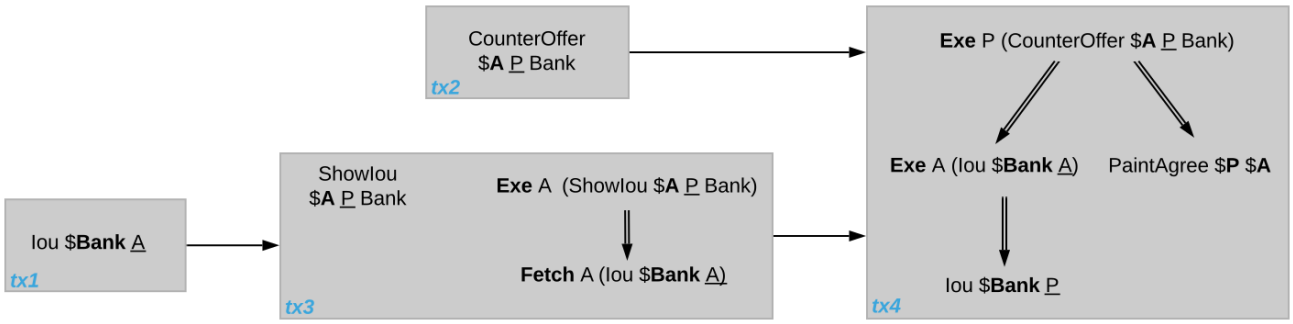


Fig. 11: Causality graph for the *counteroffer workflow with four commits*.

5.5.2.1 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.

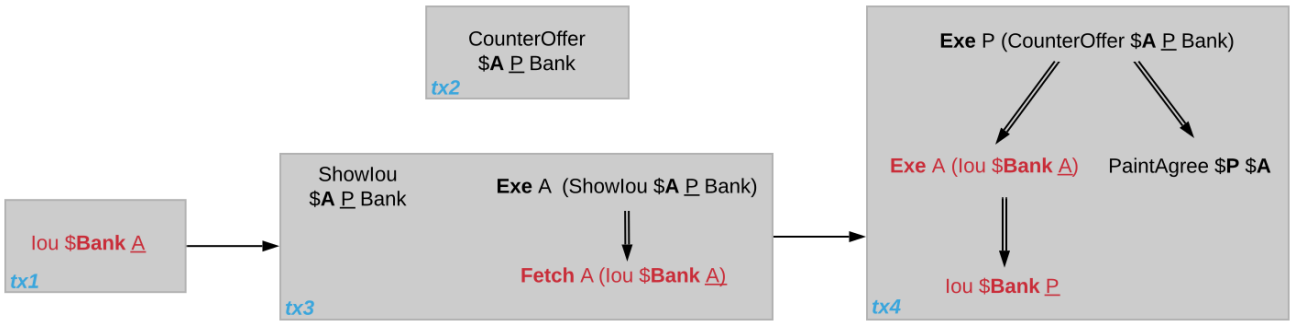
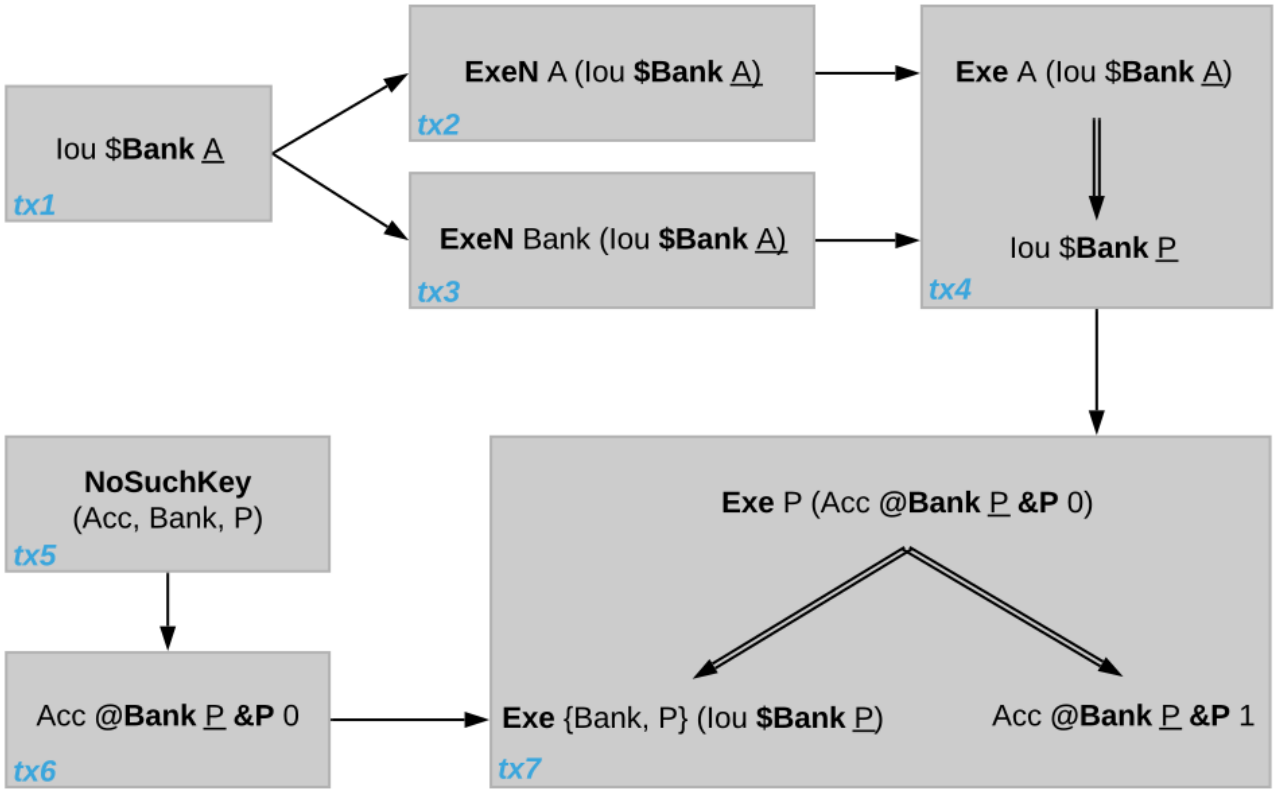


Fig. 12: Minimal consistent causality graph for the highlighted actions.

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.

5.5.2.2 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*. $reduce_X(G)$ 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 $reduce(G_L)$.

5.5.3 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.

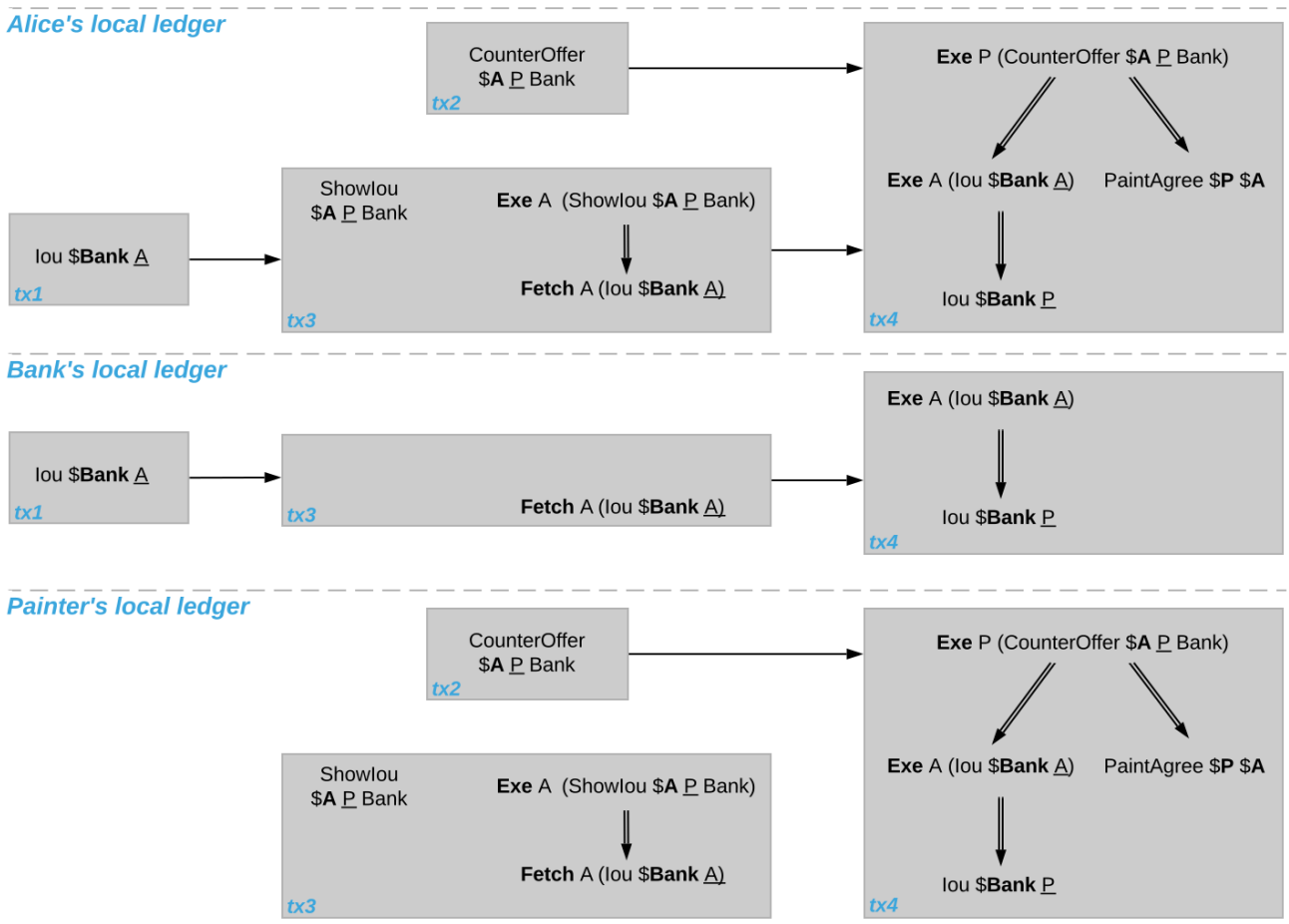


Fig. 13: Projections of the *split counteroffer causality graph*.

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*.

5.5.3.1 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.

5.5.3.2 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.

5.6 Daml Ecosystem Overview

5.6.1 Status Definitions

Throughout the documentation, we use labels to mark features of APIs not yet deemed stable. This page gives meaning to those labels.

5.6.1.1 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.

5.6.1.2 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.

5.6.1.3 Comparison of Statuses

The table below gives a concise overview of the labels used for Daml features and components.

Table 2: 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

5.6.2 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*.

5.6.2.1 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

5.6.2.2 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	

5.6.2.3 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	
Maven artifact `daml-lf-1.6-archive-java-proto`	Stable	
Maven artifact `daml-lf-1.7-archive-java-proto`	Stable	
Maven artifact `daml-lf-1.8-archive-java-proto`	Stable	
Maven artifact `daml-lf-dev-archive-java-proto`	Alpha	
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	

5.6.2.4 Developer Tools

Component / Feature	Status	Deprecated on
SDK		
Windows SDK (installer)	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	
Assistant commands to start Runtime Components: `daml json-api`, `daml trigger`, and `daml trigger-service`.	See *Runtime components*.	

continues on next page

Table 3 – continued from previous page

Component / Feature	Status	Deprecated on
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 Model Visualization (`daml damlc visual` and `daml damlc visual-web`)	Labs	
`daml doctest`	Labs	
Script		
Script Daml API	Stable	
Daml Scenario IDE integration	Stable	
Daml Script IDE integration	Stable	
Daml Script Library	See *Libraries*	

continues on next page

Table 3 – continued from previous page

Component / Feature	Status	Deprecated on
`daml test` in-memory Script and Scenario test CLI	Stable	
`daml script` *CLI to run Scripts against live ledgers.*	Stable	
`daml ledger export script` *CLI extract Daml Script from ledgers.*	Alpha	
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	

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.

5.6.3 Architecture

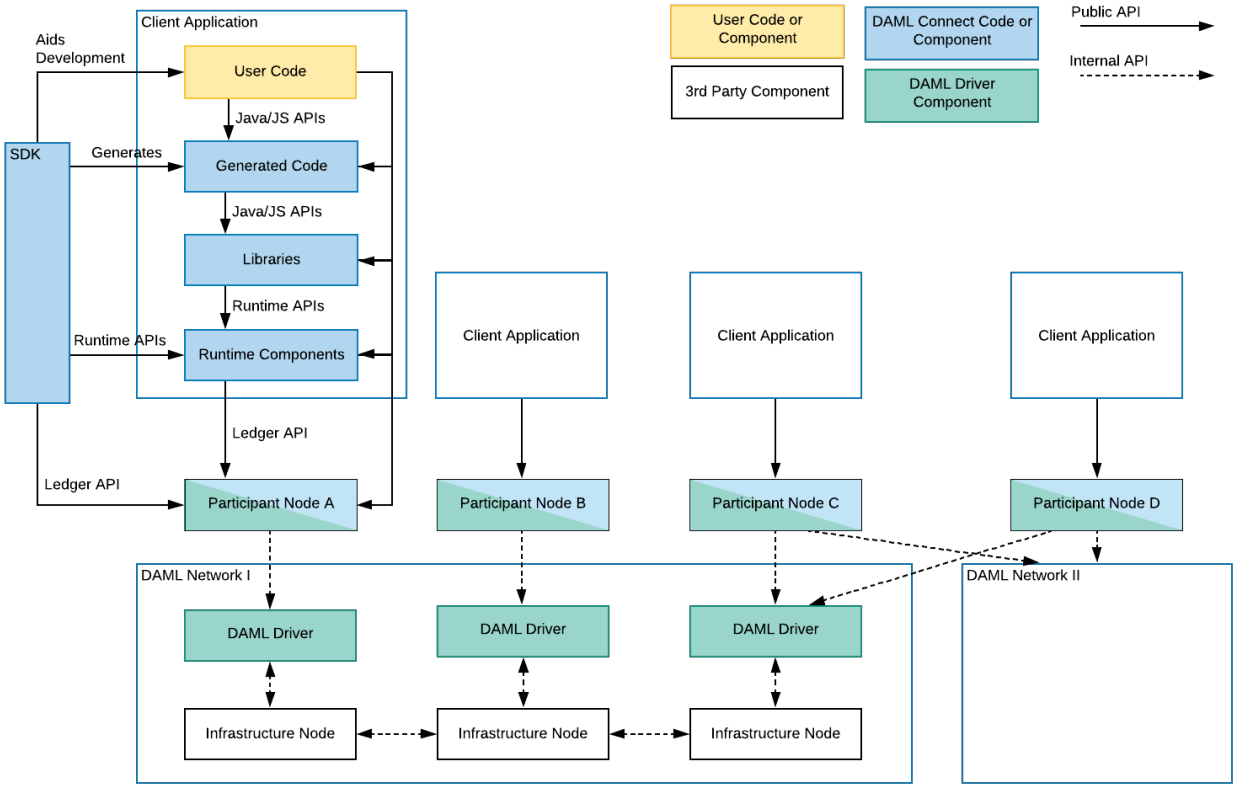
A high level view of the architecture of a Daml application or solution is helpful to make sense of how individual components, APIs and features fit into the Daml Stack.

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.

5.6.4 Daml Networks

5.6.4.1 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 WMware driver.



5.6.5 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.

5.6.6 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.

5.6.7 Daml Components

5.6.7.1 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.

5.6.7.2 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*.

5.6.7.3 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.

5.6.7.4 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*.

5.7 Releases and Versioning

5.7.1 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*.

5.7.2 Cadence

Regular snapshot releases are made every Wednesday, with additional snapshots released as needed. These releases contain Daml Components, both from the daml repository as well as some others.

Stable versions are released once a month. See *Process* below for the usual schedule. This schedule is a guide, not a guarantee: additional releases may be made, or releases may be delayed or skipped entirely.

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.

5.7.3 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.

5.7.4 Release Notes

Release notes for each release are published on the Release Notes section of the Daml Driven blog.

5.7.5 Roadmap

Once a month Digital Asset publishes a community update to accompany the announcement of the release candidate for the next release. The community update contains a section outlining the next priorities for development. You can find community updates on the Daml Driven Blog, or subscribe to the mailing list or social media profiles on https://daml.com/ to stay up to date.

5.7.6 Process

Weekly snapshot and monthly stable releases follow a regular process and schedule. The process is documented in the Daml repository so only the schedule for monthly releases is covered here.

Selecting a Release Candidate

> This is done by the Daml core engineering teams on the **first Monday of every month**.
>
> The monthly releases are time-based, not 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 on **the Thursday following the first Monday of every Month**.

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 on **the second Wednesday after the first Monday of the Month**.

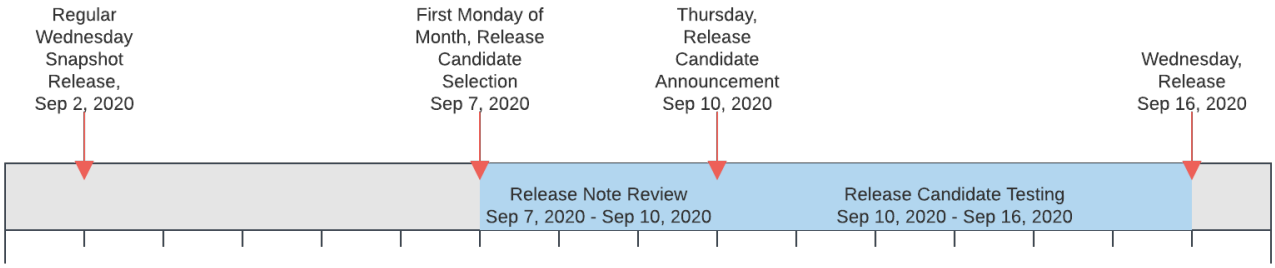


Fig. 14: The release process timeline illustrated by example of September 2020.

Chapter 6

Early Access

6.1 Ledger Export

Export is currently an *Early Access Feature in Alpha status*.

6.1.1 Introduction

Daml ledger exports read the transaction history or active contract set (ACS) from the ledger and write it to disk encoded as a *Daml Script* that will reproduce the ledger state when executed. This can be useful to migrate the history or state of a ledger from one network to another, or to replicate the ledger state locally for testing or debugging purposes.

6.1.2 Usage

The command to generate a Daml ledger export has the following form.

```
daml ledger export <format> <options>
```

Right now Daml script, specified as `script`, is the only supported export format. You can get an overview of the available command-line options using the `--help` flag as follows.

```
daml ledger export script --help
```

A full example invocation looks like this:

```
daml ledger export script --host localhost --port 6865 --party Alice --party Bob -
↪-output ../out --sdk-version 0.0.0
```

The flags `--host` and `--port` define the Daml ledger to connect to. You can omit these flags if you are invoking the command from within a Daml project with a running ledger, e.g. with a running `daml start`.

The `--party` flags define which contracts will be included in the export. In the above example only contracts visible to the parties Alice and Bob will be included in the export. Alternatively, you can set `--all-parties` to export contracts seen by all known parties. Lack of visibility of certain events may cause references to *unknown contract ids*.

The `--output` flag defines the directory prefix under which to generate the Daml project that contains the Daml script that represents the ledger export. The flag `--sdk-version` defines which Daml SDK version to configure in the generated `daml.yaml` configuration file.

By default an export will reproduce all transactions in the ledger history. The *ledger offsets* section describes how to change this behavior.

6.1.3 Output

6.1.3.1 Daml Script

The generated Daml code in `Export.daml` contains the following top-level definitions:

`type Parties` A mapping from parties in the original ledger state to parties to be used in the new reconstructed ledger state.

`lookupParty : Text -> Parties -> Party` A helper function to look up parties in the `Parties` mapping.

`allocateParties : Script Parties` A Daml script that allocates fresh parties on the ledger and returns them in a `Parties` mapping.

`type Contracts` A mapping from unknown contract ids to replacement contract ids, see *unknown contract ids*.

`lookupContract : Text -> Contracts -> ContractId a` A helper function to look up unknown contract ids in the `Contracts` mapping.

`data Args` A record that holds all arguments to the export script.

`export : Args -> Script ()` The Daml ledger export encoded as a Daml script. Given the relevant arguments this script will reproduce the ledger state when executed. You can read this script to understand the exported ledger state or history, and you can modify this script for debugging or testing purposes.

`testExport : Script ()` A Daml script that will first invoke `allocateParties` and then `export`. It will use an empty `Contracts` mapping. This can be useful to test the export in Daml studio. If your export references unknown contract ids then you may need to manually extend the `Contracts` mapping.

In most simple cases the generated Daml script will use the functions `submit` or `submitMulti` to issue ledger commands that reproduce a transaction or ACS. In some cases the generated Daml script will fall back to the more general functions `submitTree` or `submitTreeMulti`.

For example, the following generated code issues a create-and-exercise command that creates an instance of `ContractA` and exercises the choice `ChoiceA`. The function `submitTree` returns a `TransactionTree` object that captures all contracts that are created in the transaction. The `fromTree` function is then used to extract the contract ids of the `ContractB` contracts that were created by `ChoiceA`.

```
tree <- submitTree alice_0 do
  createAndExerciseCmd
    Main.ContractA with
      owner = alice_0
    Main.ChoiceA
let contractB_1_1 = fromTree tree $
      exercised @Main.ContractA "ChoiceA" $
      created @Main.ContractB
let contractB_1_2 = fromTree tree $
```

(continues on next page)

(continued from previous page)

```
      exercised @Main.ContractA "ChoiceA" $
      createdN @Main.ContractB 1
```

6.1.3.2 Arguments

Daml export will generate a default arguments file in `args.json`, which configures the export to use the same party names as in the original ledger state and to map unknown contract ids to themselves. For example:

```
{
  "contracts": {
    "001335..": "001335..."
  },
  "parties": {
    "Alice": "Alice",
    "Bob": "Bob"
  }
}
```

6.1.4 Executing the Export

The generated Daml project is configured such that `daml start` will execute the Daml export with the default arguments defined in `args.json`. Alternatively you can build and execute the generated Daml script manually using commands of the following form:

```
daml build
daml script --ledger-host localhost --ledger-port 6865 --dar .daml/dist/export-1.
↪0.0.dar --script-name Export:export --input-file args.json
```

The arguments `--ledger-host` and `--ledger-port` configure the address of the ledger and the argument `--input-file` points to a JSON file that defines the export script's arguments.

6.1.5 Ledger Offsets

By default `daml ledger export` will reproduce all transactions, as seen by the selected parties, from the beginning of the ledger history to the current end. The command-line flags `--start` and `-end` can be used to change this behavior. Both flags accept ledger offsets, either the special offsets `start` and `end`, or an arbitrary ledger offset.

`--start` Transactions up to and including the start offset will be reproduced as a sequence of create commands that reproduce the ACS as of the start offset. Later transactions will be reproduced as seen by the configured parties. In particular, `--start end` will reproduce the current ACS but no transaction history, `--start start` (the default) will reproduce the history of all transactions as seen by the configured parties.

`--end` Export transactions up to and including this end offset.

6.1.6 Unknown Contract Ids

Daml ledger export may encounter references to unknown contracts. This may occur if a contract was divulged to one of the configured parties, but the event that initially created that contract is not visible to any of the configured parties. This may also occur if a contract was archived before the configured start offset, such that it is neither part of the recreated ACS nor created in any of the exported transactions, and another live contract retains a reference to this archived contract.

In such cases Daml export will not generate commands to recreate these unknown contracts. Instead, it will generate a lookup in the `Contracts` mapping defined in the scripts arguments. You can define a mapping from unknown contract ids to replacement contract ids in the JSON input file. The default `args.json` generated by Daml ledger export will map unknown contract ids to themselves.

Note that you may submit references to non-existing contract ids to the ledger using this feature. A `fetch` on such a dangling contract id will fail.

6.1.7 Transaction Time

Daml ledger exports may fail to reproduce the ledger state or transaction history if contracts are sensitive to ledger time. You can enable the `--set-time` option to issue `setTime` commands in the generated Daml script. However, this is not supported by all ledgers.

6.1.8 Caveats

6.1.8.1 Contracts Created and Referenced in Same Transaction

Daml ledger export may fail in certain cases when it attempts to reproduce a transaction that creates a contract and then references that contract within the same transaction.

The Daml ledger API allows only a few ways in which a contract that was created in a set of commands can be referenced within the same set of commands. Namely, create-and-exercise and exercise-by-key. Choice implementations, on the other hand, are not restricted in this way.

If the configured parties only see part of a given transaction tree, then events that were originally emitted by a choice may be lifted to the root of the transaction tree. This could produce a transaction tree that cannot be replicated using the ledger API. In such cases Daml ledger export will fail.

6.2 Visualizing Daml Contracts

Visualizing Daml Contracts is currently an *Early Access Feature in Labs status*.

You can generate visual graphs for the contracts in your Daml project. To do this:

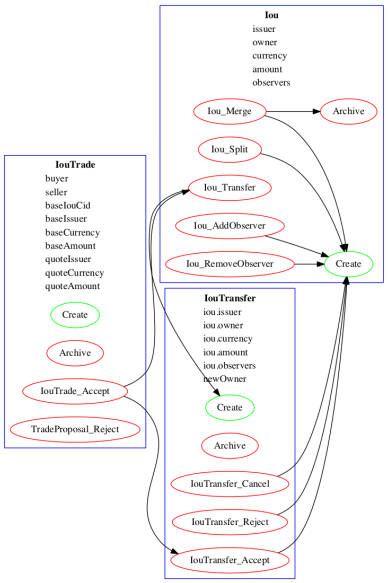
1. Install Graphviz.
2. Generate a DAR from your project by running `daml build`.
3. Generate a dot file from that DAR by running `daml damlc visual <path_to_project>/dist/<project_name.dar> --dot <project_name>.dot`
4. Generate the visual graph with Graphviz by running `dot -Tpng <project_name>.dot > <project_name>.png`

6.2.1 Example: Visualizing the Quickstart project

Here's an example visualization based on the *quickstart*. You'll need to install Graphviz to try this out.

1. Generate the dar using `daml build`
2. Generate a dot file `daml damlc visual dist/quickstart-0.0.1.dar --dot quickstart.dot`
3. Generate the visual graph with Graphviz by running `dot -Tpng quickstart.dot -o quickstart.png`

Running the above should produce an image which looks something like this:



6.2.2 Visualizing Daml Contracts - Within IDE

You can generate visual graphs from VS Code IDE. Open the daml project in VS Code and use command palette. Should reveal a new window pane with dot image. Also visual generates only the currently open daml file and its imports.

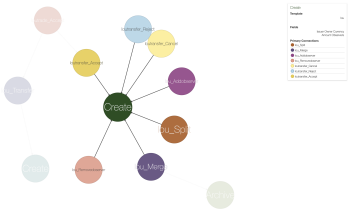
Note: You will need to install the Graphviz/dot packages as mentioned above.

6.2.3 Visualizing Daml Contracts - Interactive Graphs

This does not require any packages installed. You can generate D3 graphs for the contracts in your Daml project. To do this

1. Generate a DAR from your project by running `daml build`
2. Generate HTML file `daml damlc visual-web .daml/dist/quickstart-0.0.1.dar -o quickstart.html`

Running the above should produce an image which looks something like this:



6.3 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.

6.3.1 Interoperability examples

6.3.1.1 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.

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.

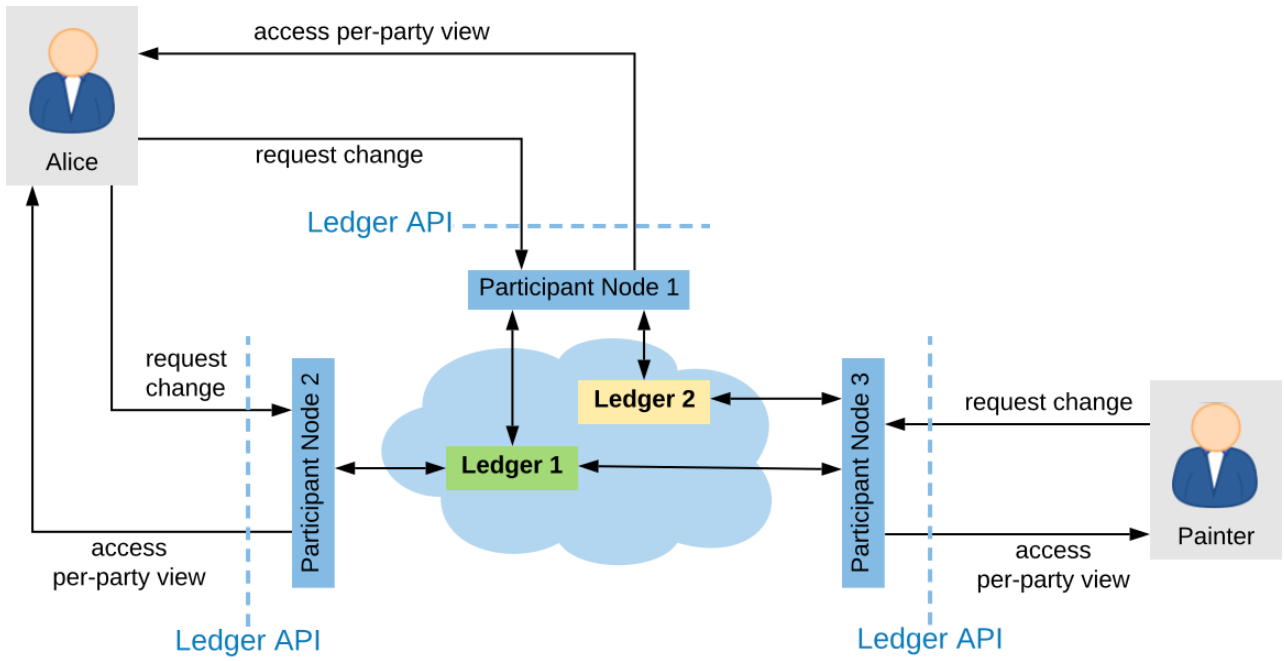


Fig. 1: Example topology with three interoperable ledgers

- Participant Node 3 hosts the painter on Ledger 1 and 2.

6.3.1.2 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.

6.3.1.3 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.

6.3.1.4 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.

6.3.2 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.

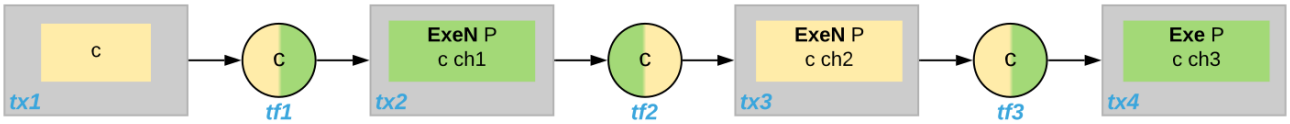


Fig. 2: Multi-Ledger causality graph with transfer actions

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.

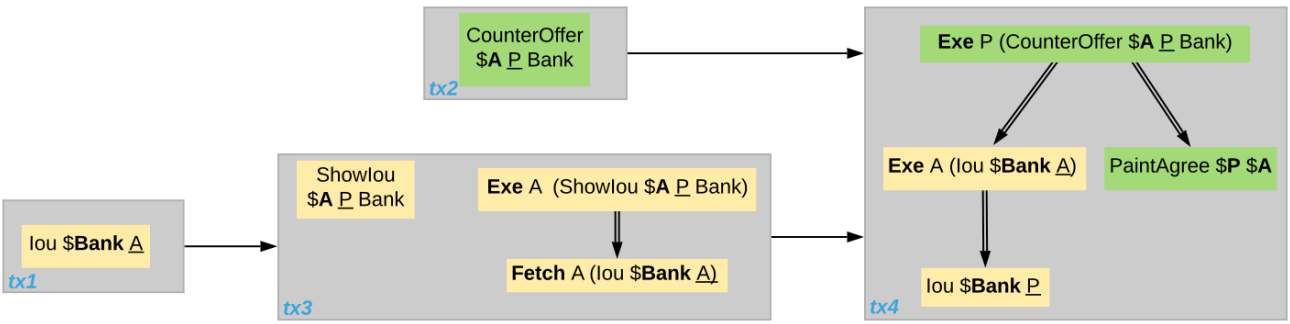


Fig. 3: Multi-Ledger causality graph for the split paint counteroffer workflow on two Daml ledgers

6.3.2.1 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 **Enter** action. This action precedes all other actions in *X*.
2. *X* contains at most one **Create** action. If so, this action precedes all other actions in *X*.
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**

6.3.2.2 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.

6.3.2.3 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.

6.3.3 Ledger-aware projection

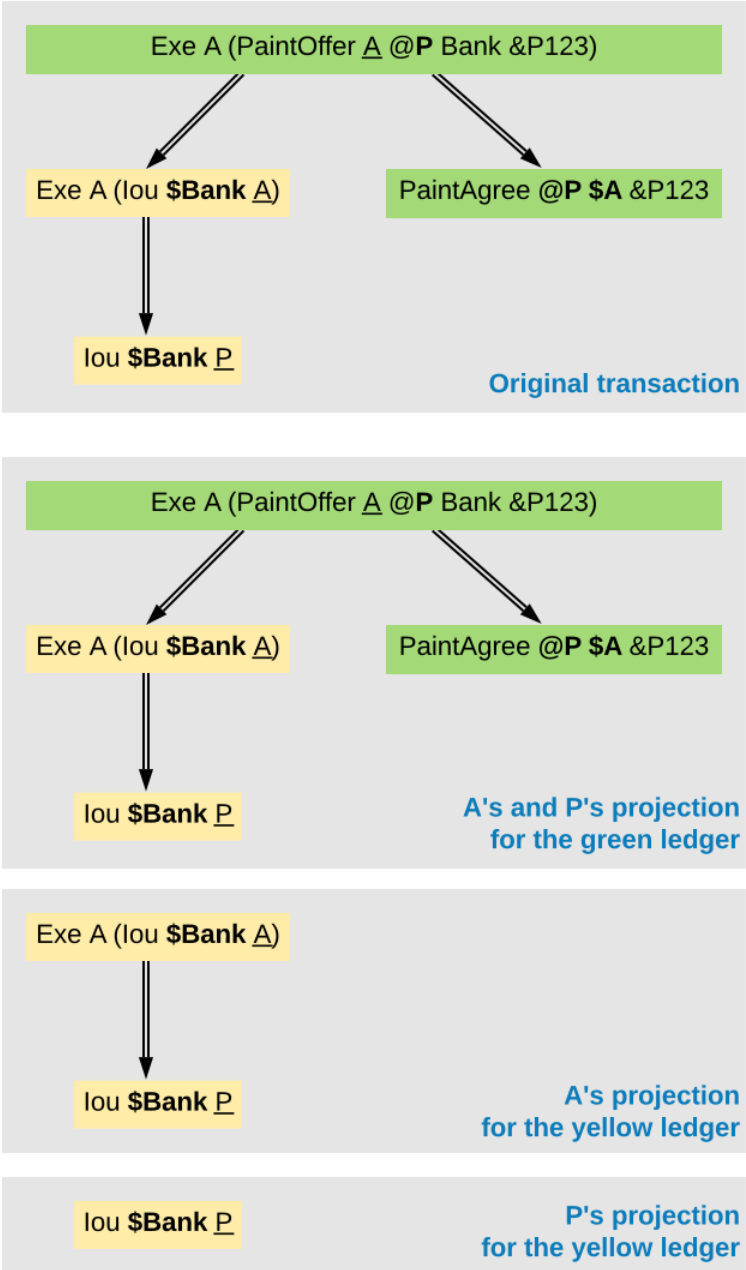
A Participant Node maintains a local ledger for each party it hosts and the Transaction Service outputs a topological sort of this local ledger. When the Participant Node hosts the party on several ledgers, this local ledger is an multi-ledger causality graph. This section defines the ledger-aware projection 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.

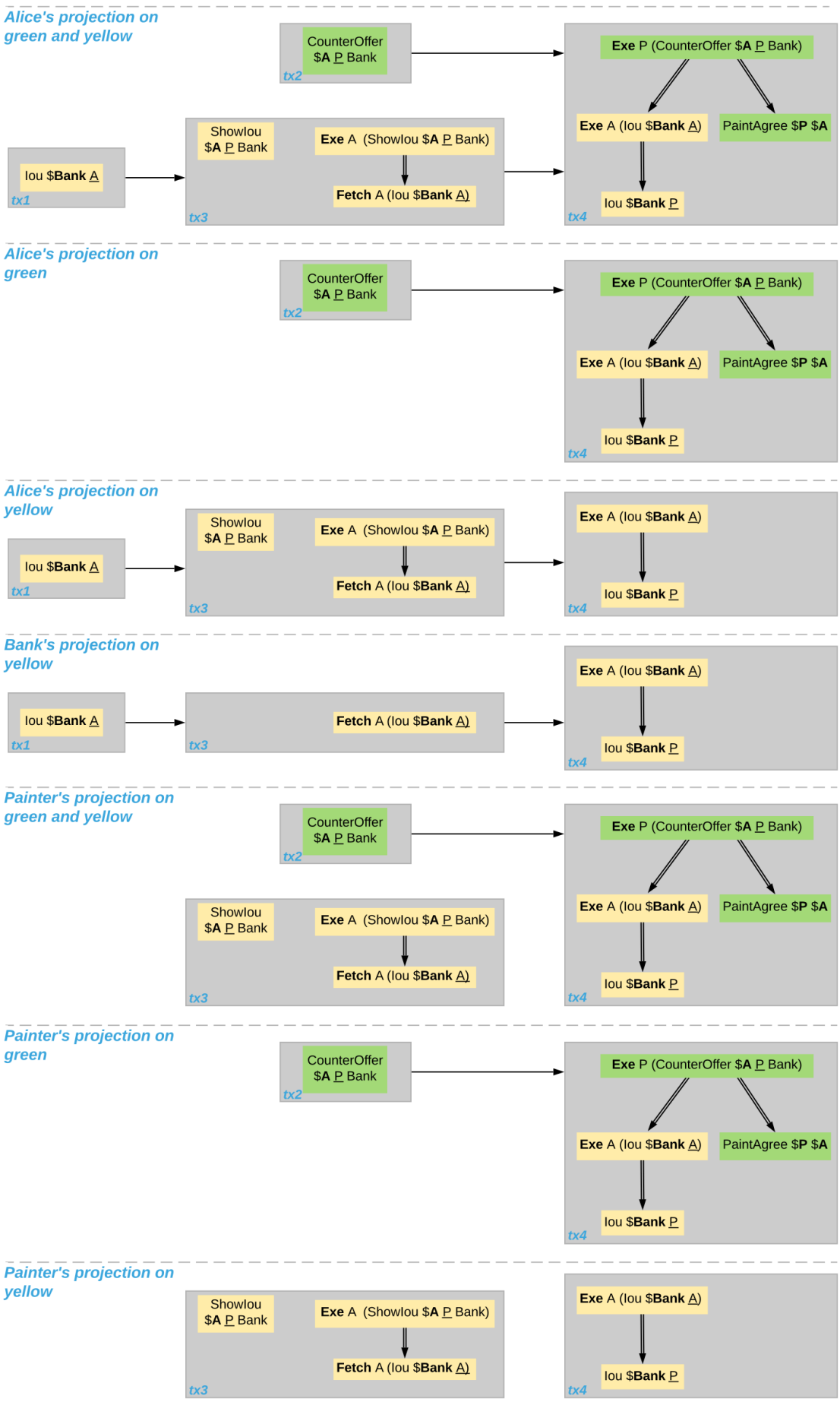
The notions of *X*-minimality and *X*-reduction extend to a party *P* on a set *Y* of ledgers accordingly.

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.

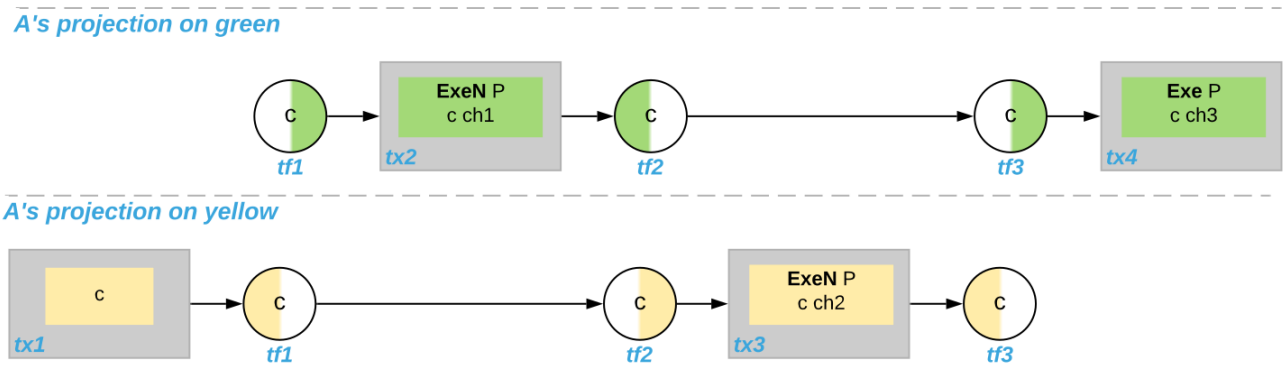
For example, the *multi-ledger causality graph for the split paint counteroffer workflow* is projected as follows.


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
Showlou $A P Bank
Exe A (Showlou $A P Bank)
Fetch A (Iou $Bank A)
tx3
Alice's projection on green
CounterOffer $A P Bank
tx2
Exe P (CounterOffer $A P Bank)
Exe A (Iou $Bank A)
PaintAgree $P $A
Iou $Bank P
tx4
Alice's projection on yellow
Iou $Bank A
tx1
Showlou $A P Bank
Exe A (Showlou $A P Bank)
Fetch A (Iou $Bank A)
tx3
Exe A (Iou $Bank A)
Iou $Bank P
tx4
Bank's projection on yellow
Iou $Bank A
tx1
Fetch A (Iou $Bank A)
tx3
Exe A (Iou $Bank A)
Iou $Bank P
tx4
Painter's projection on green and yellow
CounterOffer $A P Bank
tx2
Showlou $A P Bank
Exe A (Showlou $A P Bank)
Fetch A (Iou $Bank A)
tx3
Exe P (CounterOffer $A P Bank)
Exe A (Iou $Bank A)
PaintAgree $P $A
Iou $Bank P
tx4
Painter's projection on green
CounterOffer $A P Bank
tx2
Exe P (CounterOffer $A P Bank)
Exe A (Iou $Bank A)
PaintAgree $P $A
Iou $Bank P
tx4
Painter's projection on yellow
Showlou $A P Bank
Exe A (Showlou $A P Bank)
Fetch A (Iou $Bank A)
tx3
Exe A (Iou $Bank A)
Iou $Bank P
tx4

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.



6.3.4 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.

6.4 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.

6.4.1 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

6.4.2 Running 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.

6.4.3 Using 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();
```

6.4.4 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)

6.4.5 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.

6.5 Operating Daml

6.5.1 Daml Helm Chart

Note: This is an Early Access feature. Note that this feature does not currently work with Daml 2.0. These docs refer to and use Daml 1.18. The feature is under active development and it will soon be available for the 2.x major release series.

We provide an Early Access version of the Helm Chart for Daml Enterprise customers. This page contains documentation for that Helm chart.

6.5.1.1 Credentials

Like all Daml Enterprise components, the Helm Chart is hosted on Artifactory. To get both the Helm chart itself and the Docker images it relies on, you will need Artifactory credentials. In the rest of this document, we assume that `$ARTIFACTORY_USERNAME` refers to your Artifactory user name, whereas `$ARTIFACTORY_PASSWORD` refers to your Artifactory API key.

6.5.1.2 Installing the Helm Chart Repository

To let your local Helm installation know about the Daml Helm chart, you need to add the repository with:

```
helm repo add daml \
  https://digitalasset.jfrog.io/artifactory/connect-helm-chart \
  --username $ARTIFACTORY_USERNAME \
  --password $ARTIFACTORY_PASSWORD
```

This will install the repository as `daml`; you can then list the available versions with:

```
helm search repo --devel -l daml
```

The `--devel` flag lets Helm know that you want to list prerelease versions (in the Semver sense). To avoid any confusion as to the production-readiness of the Helm chart, while the feature is in Early Access, only prerelease versions of the Helm chart will be available.

Later on, you can update your local listing to match the Artifactory state with:

```
helm repo update
```

And you can deploy the latest prerelease version with:

```
helm install dm daml/daml-connect --devel --values values.yaml
```

where `values.yaml` is a YAML file that includes at least the `imagePullSecret` key. See the rest of this page for other options in `values.yaml`, and the Helm documentation for related Helm usage.

6.5.1.3 Setting Up the `imagePullSecret`

The Helm chart relies on the production-ready Docker images for individual components that are part of Daml Enterprise. Specifically, it expects a Kubernetes secret given as the `imagePullSecret` argument with the relevant Docker credentials in it.

Here is an example script that would load said credentials in a secret named `daml-docker-credentials`:

```
#!/usr/bin/env bash

set -euo pipefail

if [ -z ${ARTIFACTORY_PASSWORD+x} ] || [ -z ${ARTIFACTORY_USERNAME+x} ]; then
    echo "Please input information from:"
    echo "https://digitalasset.jfrog.io/ui/admin/artifactory/user_profile"
    read -p "User Profile (first.last): " USERNAME
```

(continues on next page)

(continued from previous page)

```
    read -p "API Key: " -s PASSWORD
else
    USERNAME="$ARTIFACTORY_USERNAME"
    PASSWORD="$ARTIFACTORY_PASSWORD"
fi

temp=$(mktemp)
trap "rm -f $temp" EXIT

cred=$(echo -n "$USERNAME:$PASSWORD" | base64)
jq -n --arg cred "$cred" '
  ["-daml-on-sql","-http-json","-oauth2-middleware","-trigger-service",""]
  | map({("digitalasset" + . + "-docker.jfrog.io"):{"auth":$cred}})
  | add
  | {auths: .}
  ' > $temp

kubectl create secret generic daml-docker-credentials \
        --from-file=.dockerconfigjson=$temp \
        --type=kubernetes.io/dockerconfigjson

rm -f $temp
trap - EXIT
```

Running this script with the environment variables `ARTIFACTORY_USERNAME` and `ARTIFACTORY_PASSWORD` set will result in a non-interactive deployment of the secret, which may be useful for CI environments.

6.5.1.4 Quickstart

The Helm chart is designed to let you get started quickly, using a default configuration that is decidedly **NOT MEANT FOR PRODUCTION USE**.

To get started against a development cluster, you can just run:

```
helm install dm daml/daml-connect \
     --devel \
     --set imagePullSecret=daml-docker-credentials
```

This assumes you have used the above script to setup your credentials, or otherwise created the secret `daml-docker-credentials`. It also assumes you run this command after having added the Daml Helm chart repository as explained above.

This is going to start the following:

For each of the state-keeping components (Daml driver for PostgreSQL, HTTP JSON API Service), an internal PostgreSQL database server. These are decidedly not production-ready. For a production setup, you'll need to provide your own databases here.
A fake, testing-only JWT minter to serve as the authentication server. This should be replaced with a real authentication server for production use. See the *Setting Up Auth0* section for an example of using an external authentication infrastructure.
A single instance of each of the following services: Daml driver for PostgreSQL, HTTP JSON API Service.

An nginx server exposing the `/v1` endpoints of the HTTP JSON API Service on a `NodePort` service type, for easy access from outside the Kubernetes cluster.

If you set up the Trigger Service and/or the OAuth2 Middleware (without setting the `production` flag), the reverse proxy will automatically proxy them too, and a separate PostgreSQL instance will be started for the Trigger Service. See the end of this page for details.

6.5.1.5 Production Setup

There are many options you may want to set for a production setup. See the reference at the end of this page for full details. At a minimum, though, you need to set the following:

`production=true`: By default, the Helm chart starts a number of components that are meant to give you a quick idea of what the Helm chart enables, but are most definitely not meant for production use. Specifically, this will disable the internal PostgreSQL instances, the mock auth server, and the reverse proxy.
`ledger.db`: If you want the Helm char to start a Daml driver For PostgreSQL instance for you, you need to set this. See reference section at the end of this page for details.
`ledger.host` and `ledger.port`: If you **do not** want the Helm chart to setup a Daml driver isntance for you, but instead want the components started by it to connect to an existing Ledger API server, fill in these options instead of the `ledger.db` object.
`jsonApi.db`: If you want the Helm chart to start the HTTP JSON API Service for you, you need to set this. See reference section at the end of this page for details.
`triggerService.db`: If you want the Helm chart to start the Trigger Service for you, you need to set this. See reference section at the end of this page for details.
`authUrl`: If you want the Helm chart to provide either a Daml driver for PostgreSQL or a OAuth2 Middleware instance, you will need to set this to the JWKS URL of your token provider.

If you start the Trigger Service, you will need to configure it, as well as the OAuth2 Middleware. See the required options for them in the reference section at the end of this page.

Finally, we also recommend looking at the `resources` option for each component and adjusting them to fit your particular use-case.

6.5.1.6 Log Aggregation

All processes write their logs directly to stdout. This means that log aggregation can be addressed at the Kubernetes level and does not require any specific support from the Helm chart itself. One fairly easy way to achieve this is using Filebeat, which regulary collects the logs of your containers and ingests them into *Elasticsearch <https://www.elastic.co/elasticsearch/>*, Logstash, Kafka, etc.

You can find external documentation on, how to setup *ElasticSearch* with *Filebeat* and *Kibana* for analyzing logs on your Kubernetes cluster here.

The *HTTP JSON API* component in the Helm chart produces JSON-encoded logs. Other components log as unstructured text.

6.5.1.7 Daml Metrics Options

The Daml driver for PostgreSQL instance and the HTTP JSON API instances started by the Helm chart are configured to expose Prometheus metrics on a port named `metrics`, using the appropriate annotations. This means that, if you are running a cluster-wide Prometheus instance, the relevant metrics should be collected automatically.

See each component's documentation for details on the metrics exposed:

Daml driver for PostgreSQL
JSON API metrics

6.5.1.8 Upgrading

Note: This section only makes sense with the `production` flag set to `true`.

Upgrading the Daml version should be done by uninstalling the existing Helm chart, waiting for all of the pods to stop, and then installing a higher version. Destroying all of the components is a safe operation because all of the state is stored in the provided database coordinates. There is no additional state within the components themselves.

The components are not designed for running concurrently with older versions, so it is imperative to wait for the existing Helm chart components to be completely shut down before installing the new one. Do not try to upgrade in place.

Assuming you do not change the database coordinates, you should have data continuity through the upgrade.

6.5.1.9 Backing Up

Note: This section only makes sense with the `production` flag set to `true`.

For a production setup, you should be providing the Helm chart with external database coordinates. The simplest approach here is to periodically back up those databases as a whole, just like you would any other database.

If you want to be more fine-grained, you *may* decide to **not** backup the database used by the HTTP JSON API Service instances. Note that it is imperative that you still backup the databases for the other components (Trigger Service and Daml driver for PostgreSQL) if you are running them.

If you are running the Helm chart solely for the HTTP JSON API Service (connected to an external Ledger API server), then you can eschew backing up entirely, as the database for the HTTP JSON API Service is an easy-to-reconstruct cache. This assume that, in this setup, the data store of the Ledger API server is, itself, properly backed up.

6.5.1.10 Securing Daml

The Helm chart assumes that the Kubernetes environment itself is trusted, and as such does not encrypt connections between components. Full TLS encryption between every component is not supported by the Helm chart. Individual components do support it, so if that is a requirement for you you can still set it up, though not through the Helm chart. Refer to the Secure Daml Infrastructure repository for guidance on how to set that up.

When using the Helm chart, we recommend against exposing the Ledger API gRPC endpoint outside of the cluster, and exposing the HTTP JSON API Service, Trigger Service, and OAuth2 Middleware endpoints only through an HTTP proxy. That is why the services started by the Helm chart are of type `ClusterIP`.

That proxy should either do TLS termination, or be itself behind a proxy that does, in which case all of the communications between the TLS termination endpoint must be happening on a fully trusted network.

See the *Setting Up Auth0* section for an example of setting up nginx to proxy external connections to the JSON API, Trigger Service and OAuth2 Middleware.

6.5.1.11 Helm Chart Options Reference

These options have been extracted from the Helm chart version `daml-connect-1.18.0-20211110.main.84.c297baae`.

authUrl

Type: string
Required: if either the ledger or the auth middleware is started

The JWKS endpoint used to get the public key to validate tokens. Used by the ledger and the OAuth2 Middleware.

imagePullSecret

Type: string
Required: true

The Kubernetes secret which is used for gaining access to the repository where the Daml Docker images are located.

jsonApi.create

Type: bool
Default: true
Required: false

Controls whether the HTTP JSON API Service is deployed.

jsonApi.db.host

Type: string
Required: if enabled & production

The hostname of the database server for the HTTP JSON API Service, if one is started by the Helm chart.

jsonApi.db.oracle.serviceName

Type: string
Required: if enabled & using Oracle

If the HTTP JSON API Service database is Oracle, this is used to set the Service Name.

jsonApi.db.port

Type: integer
Required: if enabled & production

The exposed port of the database server for the HTTP JSON API Service, if one is started by the Helm chart.

jsonApi.db.postgres.database

Type: string
Required: if enabled & using an external PostgreSQL

The database the HTTP JSON API Service should use when connecting to the database server.

jsonApi.db.secret

Type: string
Required: if enabled & production

The Kubernetes secret which is used for gaining access to the database. The content should have the following structure:

```
username: daml
password: s3cr3t
```

or as JSON:

```
{
    "username": "daml",
    "password": "s3cr3t"
}
```

jsonApi.db.setupSecret

Type: string
Default: none
Required: false

The HTTP JSON API Service supports a mode where the credentials used at startup (to create the database structure) are not the same as the credentials used while the application is running. This can be useful if you want to run with lower privileges, specifically without the privileges to alter table structure.

If this option is given, a separate instance of the HTTP JSON API Service will be started with `start-mode=create-only` using these credentials as a one-time job, while the regular, long-lived instances will be started with `start-mode=start-only`. If this option is **not** given, then no separate one-time job is started and regular instances are started with `start-mode=create-if-needed-and-start`.

The format of this option is the same as `jsonApi.db.secret`.

jsonApi.healthCheck

Type: string
Default: see below
Required: false

Overrides the probes for the long-lived HTTP JSON API Service instances. The current default is:

```
readinessProbe:
  httpGet:
    path: /readyz
    port: http
  initialDelaySeconds: 10
  periodSeconds: 5
startupProbe:
  httpGet:
    path: /livez
    port: http
  failureThreshold: 30
  periodSeconds: 10
livenessProbe:
  httpGet:
    path: /livez
    port: http
  initialDelaySeconds: 10
  failureThreshold: 1
  periodSeconds: 5
```

jsonApi.logLevel

Type: string
Default: info
Required: false

Sets the log level for the HTTP JSON API Service instances. Valid values are `error`, `warning`, `info`, `debug` and `trace`.

jsonApi.podAnnotations

Type: object
Default: {}
Required: false

The annotations which should be attached to the metadata of the HTTP JSON API Service pods.

jsonApi.replicaCount

Type: number
Default: 1
Required: false

Controls how many long-lived instance of the HTTP JSON API Service are started.

jsonApi.resources

Type: object
Default: see below
Required: false

Overrides the `resources` field on the HTTP JSON API Service pods. Default:

```
limits:
  cpu: "1"
  memory: "2Gi"
requests:
  cpu: "0.5"
  memory: "1Gi"
```

jsonApi.serviceAccount

Type: string
Default: null
Required: false

The service account which should be attached to the HTTP JSON API Service pods.

ledger.create

Type: bool
Default: true
Required: false

If true, the Helm chart will create a Daml driver for PostgreSQL instance. If false, you will need to provide `ledger.host` and `ledger.port` (see below).

ledger.db.host

Type: string
Required: if enabled & production

The hostname of the database server for the Daml driver for PostgreSQL, if one is started by the Helm chart.

ledger.db.port

Type: integer
Required: if enabled & production

The exposed port of the database server for the Daml driver for PostgreSQL, if one is started by the Helm chart.

ledger.db.postgres.database

Type: string
Required: if enabled & production

The database the Daml driver for PostgreSQL should use when connecting to the database server. Note that, unlike the Trigger Service and HTTP JSON API Service, the Daml driver for PostgreSQL started by the Helm chart only supports PostgreSQL database servers.

ledger.db.secret

Type: string
Required: if enabled & production

The Kubernetes secret which is used for gaining access to the database. The content should have the following structure:

```
username: daml
password: s3cr3t
```

or as JSON:

```
{
    "username": "daml",
    "password": "s3cr3t"
}
```

ledger.db.setupSecret

Type: string
Default: none
Required: false

The Daml driver for PostgreSQL supports two start modes: `--migrate-only` and `--migrate-and-start`. The long-running instance always starts with `--migrate-and-start`, but if you supply this option, the Helm chart will start a separate, one-time job with `--migrate-only`.

This can be used to supply separate credentials with table alteration privileges to the one-time job (this property), and restricted credentials with no table creation/alteration privileges to the long-running one (`ledger.db.secret`).

The structure is the same as `ledger.db.secret`.

ledger.healthCheck

Type: string
Default: see below
Required: false

Overrides the probes for the long-running Daml driver for PostgreSQL instance. Defaults:

```
readinessProbe:
  exec:
    command: ["./grpc-health-probe", "-addr=:6865" ]
  initialDelaySeconds: 5
  failureThreshold: 30
  periodSeconds: 5
livenessProbe:
  exec:
    command: ["./grpc-health-probe", "-addr=:6865" ]
  initialDelaySeconds: 10
  failureThreshold: 30
  periodSeconds: 5
```

ledger.host

Type: string
Required: if ledger.create is false

If the Helm chart should not create its own Daml driver for PostgreSQL instance (i.e. you want to connect to other components to an existing gRPC Ledger API provider), this option should be set to the hostname of the gRPC Ledger API Server to connect to.

ledger.podAnnotations

Type: object
Default: {}
Required: false

The annotations which should be attached to the metadata of the Daml driver for PostgreSQL pod.

ledger.port

Type: number
Default: 6865
Required: false

The port on which the external gRPC Ledger API Server is exposed.

ledger.resources

Type: object
Default: see below
Required: false

Overrides the `resources` field of the Daml driver for PostgreSQL pod. Defaults:

```
limits:
  cpu: "1"
  memory: "2Gi"
requests:
  cpu: "0.5"
  memory: "1Gi"
```

ledger.serviceAccount

Type: string
Default: null
Required: false

The service account which should be attached to the Daml driver for PostgreSQL pod.

oauthMiddleware.callback

Type: string
Required: if oauthMiddleware.create

The `--callback` argument given to the *OAuth 2.0 Auth Middleware* instance.

oauthMiddleware.clientId

Type: string
Required: if oauthMiddleware.create

The value of the `DAML_CLIENT_ID` environment variable needed by the *OAuth 2.0 Auth Middleware* instance.

oauthMiddleware.clientSecret

Type: string
Required: if oauthMiddleware.create

The value of the `DAML_CLIENT_SECRET` environment variable needed by the *OAuth 2.0 Auth Middleware* instance.

oauthMiddleware.create

Type: bool
Default: false
Required: false

Controls whether the OAuth2 Middleware should be deployed.

oauthMiddleware.healthCheck

Type: string
Default: see below
Required: false

Overrides the probes for the OAuth2 Auth Middleware instance. Defaults:

```
startupProbe:
  httpGet:
    path: /livez
    port: http
  failureThreshold: 30
  periodSeconds: 10
livenessProbe:
  httpGet:
    path: /livez
    port: http
  initialDelaySeconds: 10
```

(continues on next page)

(continued from previous page)

```
  failureThreshold: 1
  periodSeconds: 5
```

oauthMiddleware.oauthAuth

Type: string
Required: true

The `oauth-auth` argument given to the *OAuth 2.0 Auth Middleware* instance.

oauthMiddleware.oauthToken

Type: string
Required: true

The `oauth-token` argument given to the *OAuth 2.0 Auth Middleware* instance.

oauthMiddleware.podAnnotations

Type: object
Default: {}
Required: false

The annotations which should be attached to the metadata of the OAuth2 Auth Middleware pod.

oauthMiddleware.replicaCount

Type: number
Default: 1
Required: false

Controls how many replicas of the OAuth2 Auth Middleware are started.

oauthMiddleware.resources

Type: object
Default: see below
Required: false

Overrides the `resources` field on the OAuth2 Auth Middleware pods. Defaults:

```
limits:
  cpu: "1"
  memory: "2Gi"
requests:
  cpu: "0.5"
  memory: "1Gi"
```

oauthMiddleware.serviceAccount

Type: string
Default: not used
Required: false

The service account which should be attached to the OAuth2 Auth Middleware pods.

production

Type: string
Default: false
Required: false

If true, disables the non-production components, and marks some important options as required.

triggerService.authCallback

Type: string
Required: true

The `--auth-callback` argument passed to the *Trigger Service* instance. Note that this should be externally-reachable.

triggerService.authExternal

Type: string
Required: true

The `--auth-external` argument passed to the *Trigger Service* instance. Note that this should be externally-reachable.

triggerService.create

Type: bool
Default: false
Required: false

Controls whether a Trigger Service instance should be created.

triggerService.db.host

Type: string
Required: if enabled & production

The hostname of the database server for the Trigger Service, if one is started by the Helm chart.

triggerService.db.oracle.serviceName

Type: string
Required: if enabled & using Oracle

If the Trigger Service database is Oracle, this is used to set the Service Name.

triggerService.db.port

Type: integer
Required: if enabled & production

The exposed port of the database server for the Trigger Service, if one is started by the Helm chart.

triggerService.db.postgres.database

Type: string
Required: if enabled & using an external PostgreSQL

The database the Trigger Service should use when connecting to the database server.

triggerService.db.secret

Type: string
Required: if enabled & production

The Kubernetes secret which is used for gaining access to the database. The content should have the following structure:

```
username: daml
password: s3cr3t
```

or as JSON:

```
{
    "username": "daml",
    "password": "s3cr3t"
}
```

triggerService.db.setupSecret

Type: string
Default: null
Required: false

The Trigger Service supports an optional argument `init-db` which, when supplied, causes the Trigger Service to initialize its database structure and rthen immediately exit. If this field is set, the Helm chart will start a separate instance of the Trigger Service in this mode, as a one-time job.

This can be used to supply separate credentials with table alteration privileges to the one-time job (this property), and restricted credentials with no table creation/alteration privileges to the long-running one (`triggerService.db.secret`).

The format of this option is the same as `triggerService.db.secret`.

triggerService.healthCheck

Type: string
Default: see below
Required: false

Overrides the probes for the long-running Trigger Service instance. Defaults:

```
startupProbe:
  httpGet:
    path: /livez
    port: http
  failureThreshold: 30
  periodSeconds: 10
livenessProbe:
  httpGet:
    path: /livez
    port: http
  initialDelaySeconds: 10
  failureThreshold: 1
  periodSeconds: 5
```

triggerService.podAnnotations

Type: object
Default: {}
Required: false

The annotations which should be attached to the metadata of the Trigger Service pod.

triggerService.resources

Type: object
Default: see below
Required: false

Overrides the `resources` field of the Trigger Service pod. Defaults:

```
limits:
  cpu: "1"
  memory: "2Gi"
requests:
  cpu: "0.5"
  memory: "1Gi"
```

triggerService.serviceAccount

Type: string
Default: not used
Required: false

The service account which should be attached to the Trigger Service pod.

6.5.2 Setting Up Auth0

Note: This is an Early Access feature. Note that this feature does not currently work with Daml 2.0. These docs refer to and use Daml 1.18. The feature is under active development and it will soon be available for the 2.x major release series.

In this section, we will walk through a complete setup of an entire Daml system using Auth0 as its authentication provider.

Note: These instructions include detailed steps to be performed through the Auth0 UI, which we do not control. They have been tested on 2021-11-02. It is possible Auth0 has updated their UI since then in ways that invalidate parts of the instructions here; if you notice any discrepancy, please report it on the forum.

6.5.2.1 Authentication v. Authorization

In a complete Daml system, the Daml components only concern themselves with *authorization*: requests are accompanied by a (signed) token that *claims* a number of rights (such as the right to *act as* a given party). The Daml system will check the signature of the token, but will not perform any further verification of the claims themselves.

On the other side of the fence, the *authentication system* needs to verify a client's identity and, based on the result, provide them with an appropriate token. It also needs to record the mapping of client identity to Daml party (or parties), such that the same external identity keeps mapping to the same on-ledger party over time.

Note that we need bidirectional communication between the Daml driver and the authentication system: the authentication system needs to contact the Daml driver to allocate new parties when a new user logs in, and the Daml driver needs to contact the authentication system to fetch the public key used to verify token signatures.

In the context of this section, the authentication system is Auth0. For more information on the Daml side, see the *Authorization* page.

6.5.2.2 Prerequisites

In order to follow along this guide, you will need:

An Auth0 tenant. See Auth0 documentation for how to create one if you don't have one already. You should get a free, dev-only one when you create an Auth0 account.
A DNS (or IP address) that Auth0 can reach, and on which you (will) run a JSON API instance. This will be used to create parties. Auth0 uses a known set of IP addresses that depends on the location you chose for your tenant, so if your application is not meant to be public you can use network rules to only let requests from these IPs through.
To know the `ledgerId` your ledger self-identifies as. Refer to your specific driver's documentation for how to set the `ledgerId` value.
An application you want to deploy on your Daml system. This is not, strictly speaking, required, but the whole experience is going to be a lot less satisfying if you don't end up with something actually running on your Daml system. In this guide, we'll use the *create-daml-app* template, which supports Auth0 out-of-the-box on its UI side.

6.5.2.3 Generating Party Allocation Credentials

Since Auth0 will be in charge of requesting the allocation of parties, the first logical step is to make it generate a token that can be used to allocate parties. This may seem recursive at first, but the token used to allocate parties only needs to have the `admin` field set to `true`; it does not require any preexisting party and does not need any `actAs` or `readAs` privileges.

In Auth0 concepts, we first need to register an API. To do so, from the Auth0 Dashboard, open up the Applications -> APIs page from the menu on the left and click Create API in the top right.

You can choose any name for the API; for the purposes of this document, we'll assume this API is named `API_NAME`. The other parameters, however, are not free to set: the API identifier **has to be** `https://daml.com/ledger-api`, and the signing algorithm **has to be** RS256 (which should be selected by default). Creating the API should automatically create a Machine-to-Machine application API_NAME (Test Application) , which we will be using to generate our tokens. You can change its name to a more appropriate one; for the remainder of this document, we will assume it is called ADMIN_TOKEN_APP.

Navigate to that application's settings page (menu on the left: Applications > Applications page, then click on the application's name). This is where you can rename the application and find out about its Client ID and Client Secret, which we'll need later on.

Now that we have an API and an application, we can generate a token with the appropriate claims. In order to do that, we need to make an Auth0 Action.

In the menu on the left, navigate to Actions > Library, then click on Build Custom in the top right. You can choose an appropriate name for your action; we'll call it ADMIN_TOKEN_ACTION. Set the Trigger

field to M2M/Client-Credentials , and leave the version of Node to the recommended one. (These instructions have been tested with Node 16.)

This will open a text editor where you can add JavaScript code that will trigger on M2M (machine to machine) connections. Replace the entire text box content with:

```
exports.onExecuteCredentialsExchange = async (event, api) => {
  if (event.client.client_id === "%%ADMIN_TOKEN_ID%%") {
    api.accessToken.setCustomClaim(
      "https://daml.com/ledger-api",
      {
        "ledgerId": "%%LEDGER_ID%%",
        "participantId": null,
        "applicationId": "party-creation",
        "admin": true,
        "actAs": []
      }
    );
  }
};
```

You need to replace `%%ADMIN_TOKEN_ID%%` with the Client ID of the ADMIN_TOKEN_APP application, and `%%LEDGER_ID%%` with your actual `ledgerId` value. You can freely choose the `applicationId` value, and should set an appropriate `participantId` if your Daml driver requires it.

You then need to click on Deploy in the top right to save this Action. Despite the text on the button, this does not (yet) deploy it anywhere.

In order to actually deploy it, we need to make that Action part of a Flow. In the menu on the left, navigate through Actions > Flows, then choose Machine to Machine. Drag the ADMIN_TOKEN_ACTION (in the Custom tab) box on the right in-between the Start and Complete black circles in the middle. Click Apply. Now your Action is deployed and, should you modify it, clicking on the Deploy button *would* directly affect your live setup.

At this point you should be able to verify, using the curl command from the Quick Start tab of the M2M application, that you get a token. You should also be able to check that the token has the expected claims. You can do that by piping the result of the curl command through:

```
cat curl-result.json | jq -r '.access_token' | sed 's/.*\.\(.*\)\..*/\1/' |
↪base64 -d
```

6.5.2.4 JWKS Endpoint

In order to verify the tokens it receives, the Daml driver needs to know the public key that matches the secret key used to sign them. Daml drivers use a standard protocol for that called JWKS; in practice, this means giving the Daml driver an HTTP URL it can query to get the keys. In the case of Auth0, that URL is located at `/.well-known/jwks.json` on the tenant.

The full address is

```
https://%%AUTH0_DOMAIN%%/.well-known/jwks.json
```

You can find the value for `%%AUTH0_DOMAIN%%` in the Domain field of the settings page for the ADMIN_TOKEN_APP application (or any other application on the same tenant).

6.5.2.5 Dynamic Party Allocation

At this point, we can generate an admin token, and the Daml driver can check its signature and thus accept it. The next step is to actually allocate parties when people connect for the first time.

First, we need to create a new application, of type Single Page Web Applications . We'll be calling it LOGIN_APP. Open up the Settings tab and scroll down to Allowed Callback URLs . There, add your application's origin (scheme, domain or IP, and port) to all three of Allowed Callback URLs, Allowed Logout URLs and Allowed Web Origins. Scroll all the way down and click Save Changes .

Create a new Action (left menu > Actions > Library, top-right Build Custom button). As usual, you can choose the name; we'll call it LOGIN_ACTION. Its type should be Login / Post Login .

Replace the default code with the following JavaScript:

```
const axios = require('axios');
// only required if JSON API is behind self-signed cert
// const https = require('https');

exports.onExecutePostLogin = async (event, api) => {
  async function getParty() {
    if (event.user.app_metadata.party !== undefined) {
      return event.user.app_metadata.party;
    } else {
      const tokenResponse = await axios.request({
        "url": "%%ADMIN_TOKEN_URL%%",
        "method": "post",
        "data": {
          "client_id": "%%ADMIN_TOKEN_ID%%",
          "client_secret": "%%ADMIN_TOKEN_SECRET%%",
          "audience": "https://daml.com/ledger-api",
          "grant_type": "client_credentials"
        },
        "headers": {
          "Content-Type": "application/json",
          "Accept": "application/json"
        }
      });
      const token = tokenResponse.data.access_token;
      const partyResponse = await axios.request({
        "url": "%%ORIGIN%%/v1/parties/allocate",
        "method": "post",
        "headers": {
          "Content-Type": "application/json",
          "Accept": "application/json",
          "Authorization": "Bearer " + token
        },
        "data": {}
        // only required if JSON API is behind self-signed cert
        //, httpsAgent: new https.Agent({ rejectUnauthorized: false })
      });
      const party = partyResponse.data.result.identifier;
      api.user.setAppMetadata("party", party);

      // optional one-time setup like creating contracts etc. here

      return party;
```

(continues on next page)

(continued from previous page)

```
    }
  };
  function setToken(party, actAs = [party], readAs = [party], applicationId =
↪event.client.name) {
    api.idToken.setCustomClaim("https://daml.com/ledger-api", party);
    api.accessToken.setCustomClaim(
      "https://daml.com/ledger-api",
      {
        "ledgerId": "%%LEDGER_ID%%",
        "participantId": null,
        "applicationId": applicationId,
        "actAs": actAs,
        "readAs": readAs,
      });
  };
  if (event.client.client_id === "%%LOGIN_ID%%") {
    const party = await getParty();
    setToken(party);
  }
};
```

where you need to replace `%%LOGIN_ID%%` with the Client ID of the LOGIN_APP application; `%%ADMIN_TOKEN_URL%%`, `%%ADMIN_TOKEN_ID%%` and `%%ADMIN_TOKEN_SECRET%%` with, respectively, the URL, `client_id` and `client_secret` values that you can find on the curl example from the Quick Start of the ADMIN_TOKEN_APP application; `%%ORIGIN%%` by the domain (or IP address) and port where Auth0 can reach your JSON API instance; and `%%LEDGER_ID%%` by the `ledgerId` you're passing into your Daml driver.

Before we can click on Deploy to save (but not deploy) this snippet, we need to do one more thing. This snippet is using a library called `axios` to make HTTP calls; we need to tell Auth0 about that, so it can provision the library at runtime.

To do that, click on the little box icon to the left of code editor, then on the button Add Module that just got revealed, and type in `axios` for the name and `0.21.1` for the version. Then, click the Create button, and then the Deploy button.

Now you need to go to Actions > Flows, choose the Login flow, and drag the LOGIN_ACTION action in-between the two black circles Start and Complete.

Click Apply. You now have a working Auth0 system that automatically allocates new parties upon first login, and remembers the mapping for future logins (that happens by setting the party in the app metadata , which Auth0 persists).

Note: If you are hosting your JSON API instance behind a self-signed certificate (Auth0 absolutely requires TLS, but can be made to work with a self-signed cert), you'll need to uncomment the `https` import and the `httpsAgent` line above. The `https` module does not require extra setup (unlike the `axios` one).

6.5.2.6 Token Refresh for Trigger Service

If you want your users to be able to run triggers, you can run an instance of the Trigger Service and expose it through the same HTTP URL. Because the Trigger Service (via the Auth Middleware) will need refreshable tokens, though, we need a bit of extra setup for that to work.

The first step on that front is to actually allow our tokens to be refreshed. Go to the settings tab of the API_NAME API (menu on the left > Applications > API > API_NAME) and scroll down. Towards the bottom of the page there should be a Allow Offline Access toggle, which is off by default. Turn it on, and save.

Next, we need to create a second Machine-to-Machine Application, which we'll call OAUTH_APP, to register the OAuth2 Middleware which will refresh tokens for the Trigger service. When creating such an application, you'll be asked for its authorized APIs; select API_NAME. Once the application is created, go to its settings tab and add `%%ORIGIN%%/auth/cb` as a callback URL.

You also need to scroll all the way down to the Advanced Settings section, open the Grant Types tab, and enable Authorization Code. Don't forget to save your changes.

Finally, we need to extend our LOGIN_ACTION to respond to requests from the OAuth2 Middleware. Navigate back to the Action code (left menu > Actions > Library > Custom > click on LOGIN_ACTION) and add a second branch to the main `if` (new code starting on the line with `CHANGES START HERE`; everything before that should remain unchanged).

```
const axios = require('axios');
// only required if JSON API is behind self-signed cert
// const https = require('https');

exports.onExecutePostLogin = async (event, api) => {
  async function getParty() {
    // unchanged
  };
  function setToken(party, actAs = [party], readAs = [party], applicationId =
↪event.client.name) {
    // unchanged
  };
  if (event.client.client_id === "%%LOGIN_ID%%") {
    const party = await getParty();
    setToken(party);
    // CHANGES START HERE
  } else if (event.client.client_id === "%%OAUTH_ID%%") {
    const party = await getParty();
    const readAs = [];
    const actAs = [];
    let appId = undefined;
    event.transaction.requested_scopes.forEach(s => {
      if (s === "admin") {
        api.access.deny("Current user is not authorized for admin token.");
      } else if (s.startsWith("readAs:")) {
        const requested_read = s.slice(7);
        if (requested_read === party) {
          readAs.push(requested_read);
        } else {
          api.access.deny("Requested unauthorized readAs: " + requested_read);
        }
      } else if (s.startsWith("actAs:")) {
```

(continues on next page)

(continued from previous page)

```
        const requested_act = s.slice(6);
        if (requested_act === party) {
          actAs.push(requested_act);
        } else {
          api.access.deny("Requested unauthorized actAs: " + requested_act)
        }
      } else if (s.startsWith("applicationId:")) {
        appId = s.slice(14);
      }
    });
    setToken(party, actAs, readAs, appId);
  }
};
```

Where `%%OAUTH_ID%%` is the Client ID of the OAUTH_APP. The OAuth2 Middleware will send a request with a number of *requested scopes*; the above code shows how to walk through them as well as a simple approach to handling them. You can change this code to fit your application's requirements.

Don't forget to click on Deploy to save your changes. This time, as the Action is already part of a Flow, clicking the Deploy button really deploys the Action and there is no further action needed.

6.5.2.7 Running Your App

Preparing Your Application

You may have an application already. In that case, use that. For the purposes of illustration, here we're going to work with a modified version of `create-daml-app`.

```
daml new --template=gsg-trigger my-project
```

The next step is to build the Daml code:

```
cd my-project
daml build
daml codegen js .daml/dist/my-project-0.1.0.dar -o ui/daml.js
```

Next, we'll build our frontend code, but first we're going to make a small change to let us demonstrate interactions with the Trigger Service.

We'll need the package ID of the main DAR for the next step, so first collect it by running:

```
daml damlc inspect .daml/dist/my-project-0.1.0.dar | head -1
```

from the root of the project. In the following, we'll refer to it as `%%PACKAGE_ID%%`.

Open up `ui/src/components/MainView.tsx` and add the `Button` component to the existing imports from `semantic-ui-react`:

```
import { Container, Grid, Header, Icon, Segment, Divider, Button } from 'semantic-
↪ui-react';
```

Scroll down a little bit, and add the following code after the `USERS_END` tag (around line 18):

```
const trig = (url: string, req: object) => async () => {
  const resp = await fetch(url, req);
  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}`);
  }
}

const list = trig("/trigger/v1/triggers?party=" + username, {});
const start = trig("/trigger/v1/triggers", {
  method: "POST",
  body: JSON.stringify({
    triggerName: "%%PACKAGE_ID%%:ChatBot:autoReply",
    party: username,
    applicationId: "frontend"
  }),
  headers: {
    'Content-Type': 'application/json'
}});
```

where `%%PACKAGE_ID%%` is the package ID of the main DAR file, as explained above.

Finally, scroll down to the end of the `Grid.Column` tag, and add:

```
// ...
  </Segment>
  <Segment>
    <Button primary fluid onClick={list}>List triggers</Button>
    <Button primary fluid onClick={start}>Start autoReply</Button>
  </Segment>
</Grid.Column>
```

Now, build your frontend with (starting at the root):

```
cd ui
npm install
REACT_APP_AUTH=auth0 \
REACT_APP_AUTH0_DOMAIN=%%AUTH0_DOMAIN%% \
REACT_APP_AUTH0_CLIENT_ID=%%LOGIN_ID%% \
npm run-script build
```

As before, `%%AUTH0_DOMAIN%%` and `%%LOGIN_ID%%` need to be replaced.

Now, we need to expose the JSON API and our static files. We'll use nginx for that, but you can use any HTTP server you (and your security team) are comfortable with, as long as it can serve static files and proxy some paths.

First, create a file `nginx/nginx.conf.sh` with the following content next to your app folder, i.e. in our example `nginx` is a sibling to `my-project`.

```
#!/usr/bin/env bash

set -euo pipefail
openssl req -x509 \
            -newkey rsa:4096 \
            -keyout /etc/ssl/private/nginx-selfsigned.key \
            -out /etc/ssl/certs/nginx-selfsigned.crt \
            -days 365 \
            -nodes \
            -subj "/C=US/ST=Oregon/L=Portland/O=Company Name/OU=Org/CN=${FRONTEND_
↪IP}"
openssl dhparam -out /etc/ssl/certs/dhparam.pem 2048
cat <<NGINX_CONFIG > /etc/nginx/nginx.conf
worker_processes auto;
pid /run/nginx.pid;
events {
  worker_connections 768;
}
http {
  sendfile on;
  tcp_nopush on;
  tcp_nodelay on;
  keepalive_timeout 65;
  types_hash_max_size 2048;
  include /etc/nginx/mime.types;
  default_type application/octet-stream;
  access_log /var/log/nginx/access.log;
  error_log /var/log/nginx/error.log;
  gzip on;

  ssl_certificate /etc/ssl/certs/nginx-selfsigned.crt;
  ssl_certificate_key /etc/ssl/private/nginx-selfsigned.key;
  ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
  ssl_prefer_server_ciphers on;
  ssl_ciphers "EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH";
  ssl_ecdh_curve secp384r1;
  ssl_session_cache shared:SSL:10m;
  ssl_session_tickets off;
  ssl_stapling on;
  ssl_stapling_verify on;
  resolver 8.8.8.8 8.8.4.4 valid=300s;
  resolver_timeout 5s;
  add_header X-Frame-Options DENY;
  add_header X-Content-Type-Options nosniff;

  ssl_dhparam /etc/ssl/certs/dhparam.pem;

  server {
    listen 80;
    return 302 https://${FRONTEND_IP}\$request_uri;
  }

  server {
    listen 443 ssl http2;
    location /v1/stream {
      proxy_pass http://${JSON_IP};
```

(continues on next page)

(continued from previous page)

```
      proxy_http_version 1.1;
      proxy_set_header Upgrade \$http_upgrade;
      proxy_set_header Connection "Upgrade";
      proxy_set_header X-Forwarded-For \$proxy_add_x_forwarded_for;
    }
    location /v1 {
      proxy_pass http://${JSON_IP};
      proxy_set_header X-Forwarded-For \$proxy_add_x_forwarded_for;
    }
    location /auth/ {
      proxy_pass http://${AUTH_IP}/;
      proxy_set_header X-Forwarded-For \$proxy_add_x_forwarded_for;
    }
    location /trigger/ {
      proxy_pass http://${TRIGGER_IP}/;
      proxy_set_header X-Forwarded-For \$proxy_add_x_forwarded_for;
    }
    root /app/ui;
    index index.html;
    location / {
      # for development, uncomment proxy_pass and comment the try_files line
      #proxy_pass http://localhost:3000/;
      try_files \$uri \$uri/ =404;
    }
  }
}
NGINX_CONFIG
```

Next, create a file `nginx/Dockerfile` with this content:

```
FROM nginx:1.21.0

COPY build /app/ui
COPY nginx.conf.sh /app/nginx.conf.sh
RUN chmod +x /app/nginx.conf.sh
CMD /app/nginx.conf.sh && exec nginx -g 'daemon off;'
```

Finally, we can build the Docker container with the following, starting in the folder that contains both `nginx` and `my-project`:

```
cp -r my-project/ui/build nginx/build
cd nginx
docker build -t frontend .
```

And that's it for building the application. We now have a DAR file that is ready to be deployed to a ledger, as well as a Docker container ready to serve our frontend. All we need now is to get a Daml system up and running. We document two paths forward here: one that relies on the Helm chart included in Daml Enterprise, and a manual setup using only the Open Source SDK.

Using the Daml Helm Chart

For simplicity, we assume that you have access to a server with a public IP address that both you and Auth0 can reach. Furthermore, we assume that you have access to Daml Enterprise credentials to download the Docker images. We also assume you can create a local cluster with `minikube` on the remote machine. Finally, we assume that you have downloaded the Helm chart in a folder called `daml-connect`.

First, start a new cluster:

```
minikube start
```

Next, load up your credentials as explained in the *Daml Helm Chart* section. We assume they are loaded under the secret named `daml-docker-credentials`.

Create a file called `values.yaml` with the following content:

```yaml
imagePullSecret: daml-docker-credentials
authUrl: "https://%%AUTH0_DOMAIN%%/.well-known/jwks.json"
oauthMiddleware:
  create: true
  oauthAuth: "https://%%AUTH0_DOMAIN%%/authorize"
  oauthToken: "https://%%AUTH0_DOMAIN%%/oauth/token"
  callback: "https://%%DOMAIN%%/auth/cb"
  clientId: "%%OAUTH_ID%%"
  clientSecret: "%%OAUTH_SECRET%%"
triggerService:
  create: true
  authExternal: "https://%%DOMAIN%%/auth"
  authCallback: "https://%%DOMAIN%%/trigger/cb"
```

where, as before:

> `%%AUTH0_DOMAIN%%` is the domain of your Auth0 tenant, displayed as the Domain property of any app within the tenant.
> `%%DOMAIN%%` is the domain on which your frontend will be exposed, and in particular here the domain to which Auth0 needs to redirect after the OAuth handshake.
> `%%OAUTH_ID%%` is, as before, the OAUTH_APP application's Client ID.
> `%%OAUTH_SECRET` is the same application's Client Secret.

Assuming that you have your Artifactory credentials in the environment variables `ARTIFACTORY_USERNAME` (user name) and `ARTIFACTORY_PASSWORD` (API key), you can add the Helm repository with:

```
helm repo add daml \
  https://digitalasset.jfrog.io/artifactory/connect-helm-chart \
  --username $ARTIFACTORY_USERNAME \
  --password $ARTIFACTORY_PASSWORD
```

And now, you can deploy your cluster:

```
helm install dm daml/daml-connect --devel --values values.yaml
```

which will start the demo, non-production mode of the Helm chart. You can now start your application with:

```
PROXY="$(minikube ip):$(kubectl get svc dm-daml-connect-reverse-proxy --
↪output=json | jq '.spec.ports[0].nodePort')"
docker run -e JSON_IP=$PROXY \
           -e AUTH_IP=$PROXY/auth \
           -e TRIGGER_IP=$PROXY/trigger \
           -e FRONTEND_IP=$DOMAIN \
           --network=host \
           frontend
```

where `$DOMAIN` is assumed to be an environment variable set to the public domain on which your server is exposed. And voilà! Your application is up and running. You should be able to log in with Auth0, exchange messages, and set up an auto-reply trigger, all by connecting your browser to `https://$DOMAIN/`.

Manually Setting Up the Daml Components

For simplicity, we assume that all of the Daml components will run on a single machine (they can find each other on `localhost`) and that this machine has either a public IP or a public DNS that Auth0 can reach (hereafter assumed to be set as the `DOMAIN` env var). Furthermore, we assume that IP/DNS is what you've configured as the callback URL in the Auth0 configuration above.

Finally, we assume that you can SSH into that machine and run `daml` and `docker` commands on it.

The rest of this section happens on that remote server.

First, we need to start the Daml driver. For this example we'll use the sandbox, but with `--implicit-party-allocation false` it should behave like a production ledger (minus persistence).

```
daml sandbox --ledgerid %%LEDGER_ID%% \
             --auth-jwt-rs256-jwks https://%%AUTH0_DOMAIN%%/.well-known/jwks.json
↪\
             --implicit-party-allocation false \
             --dar .daml/dist/my-project-0.1.0.dar
```

As before, you need to replace `%%LEDGER_ID%%` with a value of your choosing (the same one you used when configuring Auth0), and `%%AUTH0_DOMAIN%%` with your Auth0 domain, which you can find as the Domain field at the top of the Settings tab for any app in the tenant.

Next, you need to start a JSON API instance.

```
cd my-project
daml json-api --ledger-port 6865 \
              --ledger-host localhost \
              --http-port 4000
```

Then, we want to start the Trigger Service and OAuth2 middleware, which we will put respectively under `/trigger` and `/auth`. First, the middleware:

```
DAML_CLIENT_ID=%%OAUTH_APP_ID%% \
DAML_CLIENT_SECRET=%%OAUTH_APP_SECRET%% \
daml oauth2-middleware \
  --address localhost \
  --http-port 5000 \
```

(continues on next page)

(continued from previous page)

```
  --oauth-auth "https://%%AUTH0_DOMAIN%%/authorize" \
  --oauth-token "https://%%AUTH0_DOMAIN%%/oauth/token" \
  --auth-jwt-rs256-jwks "https://%%AUTH0_DOMAIN%%/.well-known/jwks.json" \
  --callback %%ORIGIN%%/auth/cb
```

where, as before, you need to replace:

> `%%OAUTH_APP_ID%%` with the Client ID value you can find at the top of the settings tab for the OAUTH_APP we just created.
> `%%OAUTH_APP_SECRET%%` with the Client Secret value you can find at the top of the settings tab for the OAUTH_APP we just created.
> `%%AUTH0_DOMAIN%%` with your tenant domain.
> `%%ORIGIN%%` with the full domain-name-or-ip & port, including scheme, under which you expose your server.

Now, the trigger service:

```
daml trigger-service \
  --address localhost \
  --http-port 6000 \
  --ledger-host localhost \
  --ledger-port 6865 \
  --auth-internal http://localhost:5000 \
  --auth-external %%ORIGIN%%/auth \
  --auth-callback %%ORIGIN%%/trigger/cb \
  --dar .daml/dist/my-project-0.1.0.dar
```

where `%%ORIGIN%%` is, as per the Auth0 configuration, `https://$DOMAIN`.

And that's all the Daml components. You can now start your frontend application with:

```
docker run -e JSON_IP=localhost:4000 \
           -e AUTH_IP=localhost:5000 \
           -e TRIGGER_IP=localhost:6000 \
           -e FRONTEND_IP=$DOMAIN \
           --network=host frontend
```

This runs a production build of your frontend code. If instead you want to develop frontend code against the rest of this setup, you can uncomment the last `proxy_pass` directive in `nginx.conf.sh`, comment the `try_files` line after it, and start a reloading development server with:

```
cd ui
npm install
REACT_APP_AUTH=auth0 \
REACT_APP_AUTH0_DOMAIN=%%AUTH0_DOMAIN%% \
REACT_APP_AUTH0_CLIENT_ID=%%LOGIN_ID%% \
npm start
```