Daml SDK Documentation

Digital Asset

Version : 1.17.0-snapshot.20210915.7841.0.b4328b3d

Copyright (c) 2021 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. Install the dependencies . . . 1
  - 1.1.2 2. Install the SDK . . . 1
  - 1.1.3 Installing the Enterprise Edition . . . 1
  - 1.1.4 Next steps . . . 2
  - 1.1.5 Alternative: manual download . . . 2
- 1.2 Getting Started with Daml . . . 6
  - 1.2.1 Prerequisites . . . 6
  - 1.2.2 Running the app . . . 6
- 1.3 App Architecture . . . 10
  - 1.3.1 The Daml Model . . . 11
  - 1.3.2 TypeScript Code Generation . . . 12
  - 1.3.3 The UI . . . 12
- 1.4 Your First Feature . . . 13
  - 1.4.1 Daml Changes . . . 13
  - 1.4.2 Running the New Feature . . . 14
  - 1.4.3 Messaging UI . . . 14
  - 1.4.4 Running the updated UI . . . 15
  - 1.4.5 Next Steps . . . 17

2 Writing Daml 18
- 2.1 An introduction to Daml . . . 18
  - 2.1.1 1 Basic contracts . . . 18
  - 2.1.2 2 Testing templates using Daml Script . . . 20
  - 2.1.3 3 Data types . . . 26
  - 2.1.4 4 Transforming data using choices . . . 41
  - 2.1.5 5 Adding constraints to a contract . . . 48
  - 2.1.6 6 Parties and authority . . . 57
  - 2.1.7 7 Composing choices . . . 66
  - 2.1.8 8 Exception Handling . . . 74
  - 2.1.9 9 Working with Dependencies . . . 78
  - 2.1.10 10 Functional Programming 101 . . . 81
  - 2.1.11 11 Intro to the Daml Standard Library . . . 92
  - 2.1.12 12 Testing Daml Contracts . . . 98
- 2.2 Language reference docs . . . 102
  - 2.2.1 Overview: template structure . . . 102
  - 2.2.2 Reference: templates . . . 105

2.2.3 Reference: choices . . . 108
2.2.4 Reference: updates . . . 112
2.2.5 Reference: data types . . . 115
2.2.6 Reference: built-in functions . . . 121
2.2.7 Reference: expressions . . . 123
2.2.8 Reference: functions . . . 125
2.2.9 Reference: scenarios . . . 128
2.2.10 Reference: Daml file structure . . . 129
2.2.11 Reference: Daml packages . . . 130
2.2.12 Contract keys . . . 136
2.2.13 Exceptions . . . 145
2.3 The standard library . . . 146
2.3.1 Module Prelude . . . 146
2.3.2 Module DA.Action . . . 176
2.3.3 Module DA.Action.State . . . 177
2.3.4 Module DA.Action.State.Class . . . 178
2.3.5 Module DA.Assert . . . 179
2.3.6 Module DA.Bifunctor . . . 179
2.3.7 Module DA.BigNumeric . . . 181
2.3.8 Module DA.Date . . . 182
2.3.9 Module DA.Either . . . 184
2.3.10 Module DA.Exception . . . 185
2.3.11 Module DA.Foldable . . . 187
2.3.12 Module DA.Functor . . . 189
2.3.13 Module DA.List . . . 189
2.3.14 Module DA.List.BuiltinOrder . . . 193
2.3.15 Module DA.List.Total . . . 195
2.3.16 Module DA.Logic . . . 195
2.3.17 Module DA.Map . . . 197
2.3.18 Module DA.Math . . . 199
2.3.19 Module DA.Monoid . . . 200
2.3.20 Module DA.Next.Map . . . 201
2.3.21 Module DA.Next.Set . . . 203
2.3.22 Module DA.NonEmpty . . . 204
2.3.23 Module DA.NonEmpty.Types . . . 205
2.3.24 Module DA.Numeric . . . 206
2.3.25 Module DA.Optional . . . 207
2.3.26 Module DA.Optional.Total . . . 207
2.3.27 Module DA.Record . . . 208
2.3.28 Module DA.Semigroup . . . 209
2.3.29 Module DA.Set . . . 209
2.3.30 Module DA.Stack . . . 211
2.3.31 Module DA.Text . . . 212
2.3.32 Module DA.TextMap . . . 215
2.3.33 Module DA.Time . . . 216
2.3.34 Module DA.Traversable . . . 217
2.3.35 Module DA.Tuple . . . 218
2.3.36 Module DA.Validation . . . 218
2.4 Troubleshooting . . . 219
2.4.1 Error: <X> is not authorized to commit an update . . . 219
2.4.2 Error Argument is not of serializable type . . . 220

2.4.3 Modeling questions . . . 220
2.4.4 Testing questions . . . 222
2.5 Good design patterns . . . 223
2.5.1 Initiate and Accept . . . 223
2.5.2 Multiple party agreement . . . 225
2.5.3 Delegation . . . 227
2.5.4 Authorization . . . 230
2.5.5 Locking . . . 232
2.5.6 Diagram legends . . . 240

3 Building applications **242**
3.1 Application architecture . . . 242
3.1.1 Backend . . . 244
3.1.2 Frontend . . . 244
3.1.3 Authorization . . . 244
3.1.4 Developer workflow . . . 245
3.2 JavaScript Client Libraries . . . 248
3.2.1 JavaScript Code Generator . . . 248
3.2.2 @daml/react . . . 253
3.2.3 @daml/ledger . . . 253
3.2.4 @daml/types . . . 253
3.2.5 Testing Your Web App . . . 253
3.3 HTTP JSON API Service . . . 256
3.3.1 Daml-LF JSON Encoding . . . 256
3.3.2 Query language . . . 263
3.3.3 Metrics . . . 265
3.3.4 Running the JSON API . . . 267
3.3.5 HTTP Status Codes . . . 271
3.3.6 Create a new Contract . . . 272
3.3.7 Creating a Contract with a Command ID . . . 274
3.3.8 Exercise by Contract ID . . . 274
3.3.9 Exercise by Contract Key . . . 276
3.3.10 Create and Exercise in the Same Transaction . . . 277
3.3.11 Fetch Contract by Contract ID . . . 279
3.3.12 Fetch Contract by Key . . . 280
3.3.13 Get all Active Contracts . . . 281
3.3.14 Get all Active Contracts Matching a Given Query . . . 282
3.3.15 Fetch Parties by Identifiers . . . 284
3.3.16 Fetch All Known Parties . . . 285
3.3.17 Allocate a New Party . . . 285
3.3.18 List All DALF Packages . . . 286
3.3.19 Download a DALF Package . . . 286
3.3.20 Upload a DAR File . . . 287
3.3.21 Streaming API . . . 288
3.3.22 Healthcheck Endpoints . . . 294
3.4 Daml Script . . . 294
3.4.1 Daml Script Library . . . 294
3.4.2 Usage . . . 298
3.4.3 Using Daml Script for Ledger Initialization . . . 301
3.4.4 Using Daml Script in Distributed Topologies . . . 302
3.4.5 Running Daml Script against Ledgers with Authorization . . . 303

3.4.6 Running Daml Script against the HTTP JSON API . . . 303
3.5 Daml REPL . . . 304
3.5.1 Usage . . . 304
3.5.2 What is in scope at the prompt? . . . 305
3.5.3 Using Daml REPL without a Ledger . . . 306
3.5.4 Connecting via TLS . . . 306
3.5.5 Connection to a Ledger with Authorization . . . 306
3.5.6 Using Daml REPL to convert to JSON . . . 306
3.6 Upgrading and Extending Daml applications . . . 306
3.6.1 Extending Daml applications . . . 306
3.6.2 Upgrading Daml applications . . . 308
3.6.3 Automating the Upgrade Process . . . 313
3.7 Authorization . . . 316
3.7.1 Introduction . . . 316
3.7.2 Access tokens and claims . . . 317
3.7.3 Getting access tokens . . . 318
3.7.4 Using access tokens . . . 318
3.8 The Ledger API . . . 318
3.8.1 The Ledger API services . . . 319
3.8.2 gRPC . . . 327
3.8.3 Ledger API Reference . . . 328
3.8.4 How Daml types are translated to protobuf . . . 366
3.8.5 How Daml types are translated to Daml-LF . . . 373
3.8.6 Java bindings . . . 378
3.8.7 Scala bindings . . . 408
3.8.8 Node.js bindings . . . 412
3.8.9 Creating your own bindings . . . 412
3.8.10 What's in the Ledger API . . . 415
3.8.11 Daml-LF . . . 416

4 Deploying to Daml ledgers **417**
4.1 Overview of Daml ledgers . . . 417
4.1.1 Commercial Integrations . . . 417
4.1.2 Open Source Integrations . . . 417
4.1.3 Daml Ledgers in Development . . . 417
4.2 Deploying to a generic Daml ledger . . . 418
4.2.1 Connecting via TLS . . . 419
4.2.2 Configuring Request Timeouts . . . 419
4.3 Daml Ledger Topologies . . . 419
4.3.1 Global State Topologies . . . 419
4.3.2 Partitioned Ledger Topologies . . . 423

5 Operating Daml **424**
5.1 Daml Participant pruning . . . 424
5.1.1 Impacts on Daml applications . . . 424
5.1.2 How the Daml Ledger API is affected . . . 425
5.1.3 Other limitations . . . 425
5.1.4 How Pruning affects Index DB administration . . . 425
5.1.5 Determining a suitable pruning offset . . . 426

6 Developer Tools **427**
6.1 Daml Assistant (`daml`) . . . 427

6.1.1 Full help for commands . . . 427
6.1.2 Configuration files . . . 427
6.1.3 Building Daml projects . . . 430
6.1.4 Managing releases . . . 431
6.1.5 Terminal Command Completion . . . 431
6.1.6 Running Commands outside of the Project Directory . . . 431
6.2 Daml Studio . . . 432
6.2.1 Installing . . . 432
6.2.2 Creating your first Daml file . . . 432
6.2.3 Supported features . . . 433
6.2.4 Common scenario errors . . . 437
6.2.5 Working with multiple packages . . . 440
6.3 Daml Sandbox . . . 441
6.3.1 Contract Identifier Generation . . . 442
6.3.2 Running with persistence . . . 442
6.3.3 Running with authentication . . . 443
6.3.4 Running with TLS . . . 444
6.3.5 Command-line reference . . . 445
6.3.6 Metrics . . . 445
6.4 Navigator . . . 451
6.4.1 Navigator functionality . . . 452
6.4.2 Installing and starting Navigator . . . 452
6.4.3 Choosing a party / changing the party . . . 452
6.4.4 Logging out . . . 453
6.4.5 Viewing templates or contracts . . . 453
6.4.6 Using Navigator . . . 456
6.4.7 Authorizing Navigator . . . 459
6.4.8 Advanced usage . . . 459
6.5 Daml codegen . . . 461
6.5.1 Introduction . . . 461
6.5.2 Running the Daml codegen . . . 461
6.6 Daml Profiler . . . 463
6.6.1 Usage . . . 463
6.6.2 Caveats . . . 464

7 Background concepts **465**
7.1 Glossary of concepts . . . 465
7.1.1 Daml . . . 465
7.1.2 Developer tools . . . 469
7.1.3 Building applications . . . 470
7.1.4 General concepts . . . 472
7.2 Daml Ledger Model . . . 472
7.2.1 Structure . . . 473
7.2.2 Integrity . . . 480
7.2.3 Privacy . . . 492
7.2.4 Daml: Defining Contract Models Compactly . . . 500
7.2.5 Exceptions . . . 501
7.3 Identity and Package Management . . . 507
7.3.1 Identity Management . . . 508
7.3.2 Package Management . . . 509
7.4 Time . . . 511

7.4.1 Ledger time . . . 511
7.4.2 Record time . . . 511
7.4.3 Guarantees . . . 511
7.4.4 Ledger time model . . . 511
7.4.5 Assigning ledger time . . . 512
7.5 Causality and Local Ledgers . . . 512
7.5.1 Causality examples . . . 512
7.5.2 Causality graphs . . . 515
7.5.3 Local ledgers . . . 519

8 Examples **523**

9 Early Access Features **524**
9.1 Extractor . . . 524
9.1.1 Introduction . . . 524
9.1.2 Setting up . . . 524
9.1.3 Trying it out . . . 525
9.1.4 Running the Extractor . . . 525
9.1.5 Connecting the Extractor to a ledger . . . 525
9.1.6 Connecting to your database . . . 525
9.1.7 Authorize Extractor . . . 526
9.1.8 Full list of options . . . 526
9.1.9 Output format . . . 527
9.1.10 Transactions . . . 528
9.1.11 Contracts . . . 528
9.1.12 Exercises . . . 529
9.1.13 JSON format . . . 530
9.1.14 Examples of output . . . 530
9.1.15 Dealing with schema evolution . . . 530
9.1.16 Logging . . . 532
9.1.17 Continuity . . . 533
9.1.18 Fault tolerance . . . 533
9.1.19 Troubleshooting . . . 533
9.2 Daml Integration Kit . . . 534
9.2.1 Ledger API Test Tool . . . 534
9.2.2 Daml Integration Kit status and roadmap . . . 538
9.2.3 Implementing your own Daml Ledger . . . 539
9.2.4 Deploying a Daml Ledger . . . 542
9.2.5 Testing a Daml Ledger . . . 543
9.2.6 Benchmarking a Daml Ledger . . . 543
9.3 Daml Triggers - Off-Ledger Automation in Daml . . . 543
9.3.1 Daml Trigger Library . . . 543
9.3.2 How To Think About Triggers . . . 555
9.3.3 Sample Trigger . . . 555
9.3.4 Daml Trigger Basics . . . 556
9.3.5 Running a No-Op Trigger . . . 557
9.3.6 Diversion: Updating `Message` . . . 557
9.3.7 AutoReply . . . 557
9.3.8 Command Deduplication . . . 559
9.3.9 Authentication . . . 560
9.3.10 When not to use Daml triggers . . . 560
9.4 Visualizing Daml Contracts . . . 560

9.4.1 Example: Visualizing the Quickstart project . . . 560
9.4.2 Visualizing Daml Contracts - Within IDE . . . 561
9.4.3 Visualizing Daml Contracts - Interactive Graphs . . . 561
9.5 Ledger Interoperability . . . 561
9.5.1 Interoperability examples . . . 562
9.5.2 Multi-ledger causality graphs . . . 564
9.5.3 Ledger-aware projection . . . 568
9.5.4 Ledger API ordering guarantees . . . 572

10 Daml Ecosystem **574**
10.1 Daml Ecosystem Overview . . . 574
10.1.1 Status Definitions . . . 574
10.1.2 Feature and Component Statuses . . . 577
10.1.3 Architecture . . . 581
10.2 Releases and Versioning . . . 582
10.2.1 Versioning . . . 583
10.2.2 Cadence . . . 583
10.2.3 Support Duration . . . 583
10.2.4 Release Notes . . . 583
10.2.5 Roadmap . . . 583
10.2.6 Process . . . 583
10.3 Portability, Compatibility, and Support Durations . . . 584
10.3.1 Ledger API Compatibility: Application Portability . . . 585
10.3.2 Driver and Participant Compatibility: Network Upgradeability . . . 586
10.3.3 SDK, Runtime Component, and Library Compatibility: Daml Connect Upgradeability . . . 586
10.3.4 Ledger API Support Duration . . . 586
10.4 Getting Help . . . 586
10.4.1 Support expectations . . . 587

Chapter 1

Getting started

1.1 Installing the SDK

1.1.1 1. Install the dependencies

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

You need to install:

1. Visual Studio Code.
2. JDK 8 or greater. If you don't already have a JDK installed, try Eclipse Adoptium.
   As part of the installation process you might need to set up the `JAVA_HOME` variable. You can find here the instructions on how to do it on *Windows,macOS, and Linux*.

1.1.2 2. Install the SDK

1.1.2.1 Windows 10

Download and run the installer, which will install Daml and set up your PATH.

1.1.2.2 Mac and Linux

To install the SDK on Mac or 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.

1.1.3 Installing the Enterprise Edition

If you have a license for the enterprise edition of Daml Connect, 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 enterprise edition, use `daml install --force VERSION`.

1.1.4 Next steps

Follow the *getting started guide*.
Use `daml --help` to see all the commands that the Daml assistant (`daml`) provides.
If the `daml` command is not available in your terminal after logging out and logging in again, you need to set the `PATH` environment variable manually. You can find instructions on how to do this *here*.
If you run into any other problems, you can use the *support page* to get in touch with us.

1.1.5 Alternative: manual download

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*.

1.1.5.1 Setting JAVA_HOME and PATH variables

Windows

We'll explain here how to set up `JAVA_HOME` and `PATH` variables on Windows.

Setting the JAVA_HOME variable

1. Open `Search` and type advanced system settings and hit `Enter`.
2. Find the `Advanced` tab and click on the `Environment Variables`.
3. In the `System variables` section click on `New` if you want to set `JAVA_HOME` system wide. To set `JAVA_HOME` for a single user click on `New` under `User variables`.
4. In the opened modal window for `Variable name` type `JAVA_HOME` and for the `Variable value` set the path to the JDK installation. Click OK once you're done.
5. Click OK and click Apply to apply the changes.

Setting the PATH variable

If you have downloaded and installed the SDK using our Windows installer your `PATH` variable is already set up.

Mac OS

First, you need to figure out whether you are running Bash or zsh. To do that, 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. We provide instructions for both, but you only need to follow the instructions for the one you are using.

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, please come and ask on the Daml forum and we will be happy to help.

Open a terminal and run the following commands. Typos are a big problem here so copy/paste one line at a time if possible. None of these should produce any output on success. If you are running **bash**, run:

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

If you are running **zsh**, run:

```
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 a 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, please contact us on the Daml forum and we will be happy to help.

Linux

We'll explain here how to set up `JAVA_HOME` and `PATH` variables on Linux for `bash`.

Setting the JAVA_HOME variable

Java should be installed typically 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 you and 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, please follow the instructions below to verify that everything was set up correctly.

Please 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`.

Next, please 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 Connect 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 release 1.4.0, you would download the files `daml-sdk-1.4.0-macos.tar.gz` and `daml-sdk-1.4.0-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 1.4.0 on macOS, you should have both files in the current directory and run:

```
gpg --verify daml-sdk-1.4.0-macos.tar.gz.asc
```

and that should give you a result that looks like:

```
gpg: assuming signed data in 'daml-sdk-1.4.0-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-1.4.0-macos.tar.gz
cd sdk-1.4.0
./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
```

(continues on next page)

(continued from previous page)

```
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. We do this through the example of a simple social networking application, showing you three things:

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*)

We do not aim to be comprehensive in all Daml concepts and tools (covered in *Writing Daml*) or in all deployment options (see *Deploying*). **For a quick overview of the most important Daml concepts used in this tutorial open** the Daml cheat-sheet **in a separate tab**. The goal is that by the end of this tutorial, you'll have a good idea of the following:

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

With that, let's get started!

1.2.1 Prerequisites

Please make sure that you have the Daml Connect SDK, Java 8 or higher, and Visual Studio Code (the only supported IDE) installed as per instructions from our *Installing the SDK* page.

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 `12.22`; 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

We'll start by getting the app up and running, and then explain the different components which we will later extend.

First off, open a terminal, change to 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`.

Change to the new folder:

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

We can now run the app in two steps. You'll need two terminal windows running for this. 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.

You will know that 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.

In a 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 (it's worth it!). You should see `success Saved lockfile.` in the output if everything worked as expected.

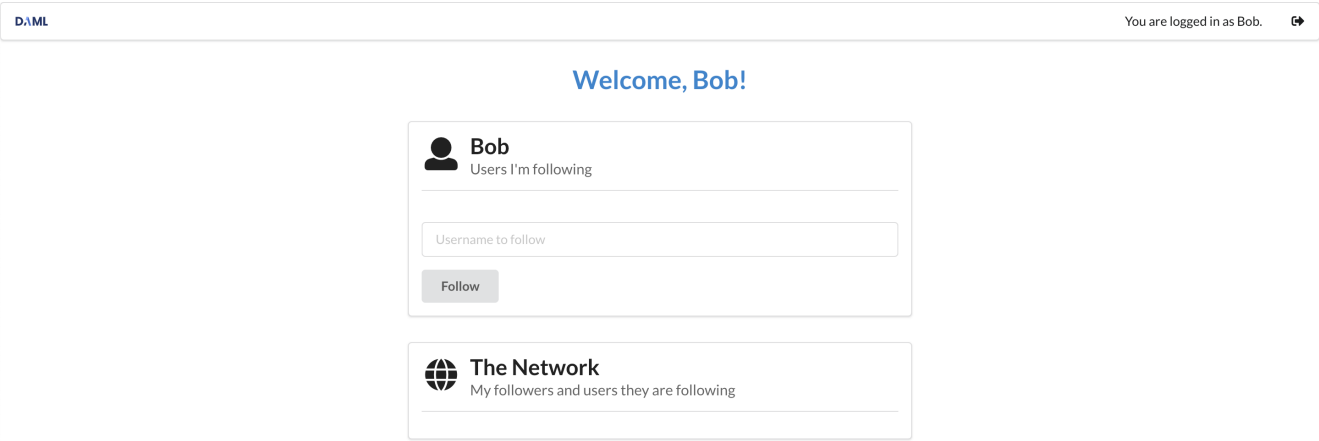
Now you can start the UI with:

```
npm start
```

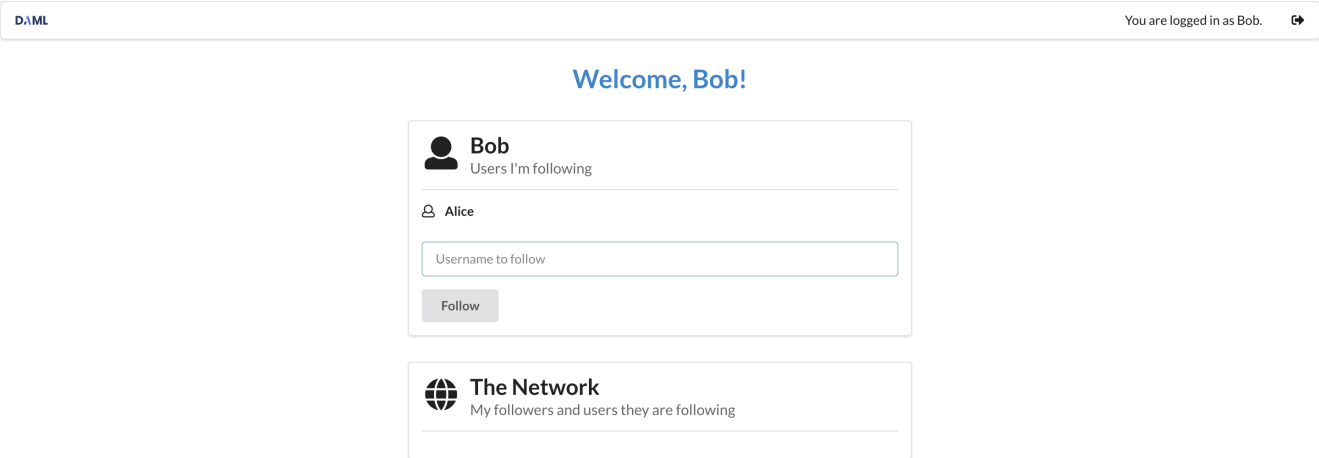
This starts the web UI connected to the running Sandbox and JSON API server. The command should automatically open a window in your default browser at http://localhost:3000. Once the web UI has been compiled and started, you should see `Compiled successfully!` in your terminal. If it doesn't, just open that link 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 of this app, there is no password or sign-up required. First enter your name and click *Log in*.

You should see the main screen with two panels. One for the users you are following and one for your followers. Initially these are both empty as you are not following anyone and you don't have any

followers! Go ahead and start following users by typing their usernames in the text box and clicking on the *Follow* button in the top panel.

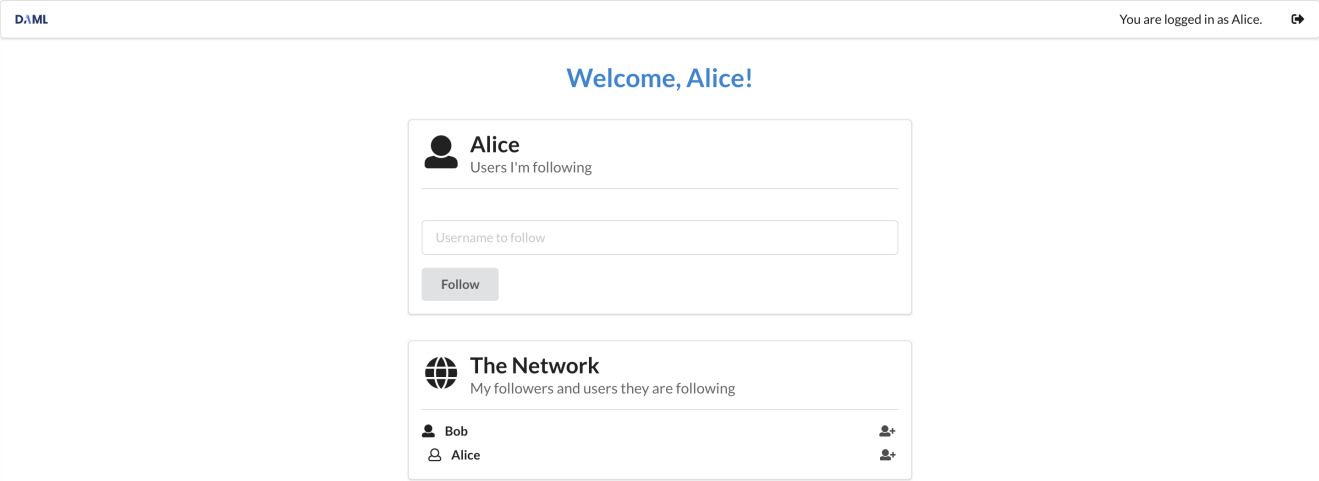


You'll notice that the users you just started following appear in the *Following* panel. However they do *not* yet appear in the *Network* panel. This is either because they have not signed up and are not parties on the ledger or they have not yet started following you. This social network is similar to Twitter and Instagram, where by following someone, say Alice, you make yourself visible to her but not vice versa. We will see how we encode this in Daml in the next section.

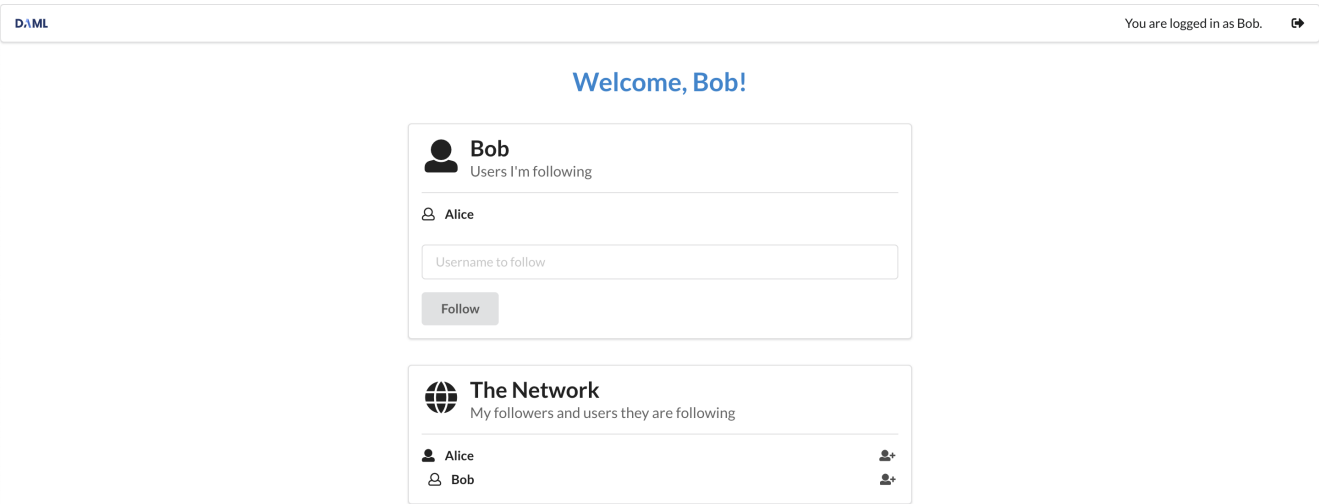


To make this relationship reciprocal, open a new browser window/tab at http://localhost:3000. (Having separate windows/tabs allows you to see both you and the screen of the user you are following at the same time.) Once you log in as the user you are following - Alice, you'll notice your name in her

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. Just 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*.

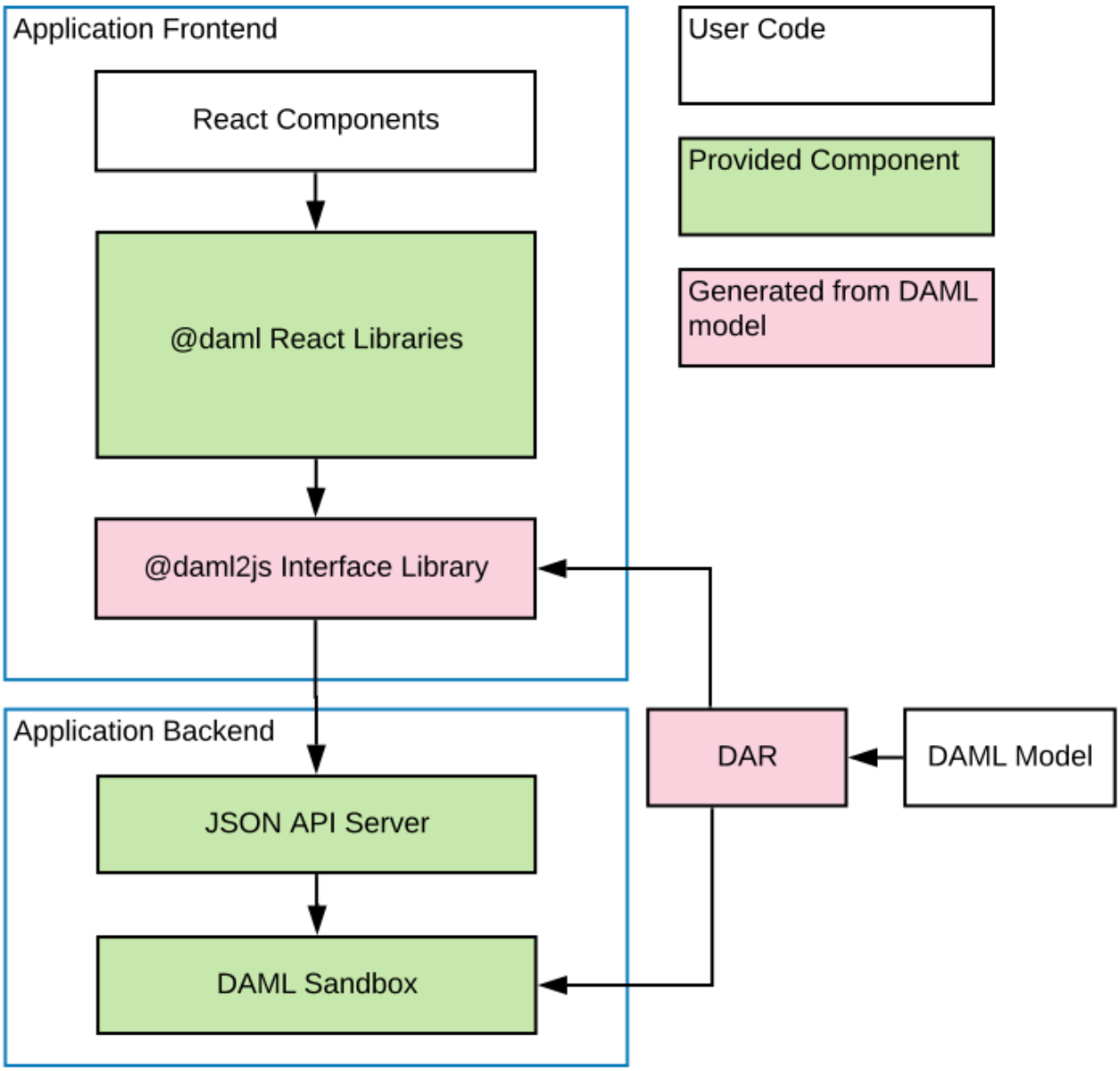
1.3 App Architecture

In this section we'll look at the different components of our social network app. The goal is to familiarize you enough to feel comfortable extending the code with a new feature in the next section. There are two main components:

the Daml model and
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 Connect - just 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.

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.

Let's see what the `signatory` and `observer` clauses mean in our app more concretely. A user Alice can see another user 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.

Here we see 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 we must answer these questions upfront, as they fundamentally change the design of an application.

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

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).

This information should be enough for understanding how choices work in this guide. More detailed information on choices can be found in *our docs*.

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.

In order 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.

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 having to thread it through manually. We take advantage of hooks in particular to share ledger state across components. We use custom *Daml React hooks* to query the ledger for contracts, create new contracts, and exercise choices. This is the library you will be using the most 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, then 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.

The `useParty` hook simply 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. This means that 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.

[1] FYI 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*.

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 gets triggered when you click the icon next to a user's name in the *Network* panel.

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

Let's dive into implementing a new feature for our social network app. This will give us a better idea how to develop Daml applications using our template.

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, respecting *authorization* and *privacy*. This means:

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

We will see that Daml lets us implement these guarantees in a direct and intuitive way.

There are three parts to building and running the messaging feature:

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

As mentioned in the *architecture* section, the Daml code defines the *data* and *workflow* of the application. The workflow aspect 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.

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 the fact 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.*

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.

1.4.2 Running the New Feature

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 at the beginning of this *Getting Started with Daml* guide, Daml Sandbox uses an in-memory store, which means it loses its state when stopped or restarted. That means that all user data and follower relationships are lost.

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

1.4.3 Messaging UI

The UI for messaging will consist of a new *Messages* panel in addition to the *Follow* and *Network* panel. This panel will have 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 will implement each part as a React component, which we'll name `MessageList` and `MessageEdit` respectively. Let's start with the simpler `MessageList`.

1.4.3.1 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. You should copy this into a new `MessageList.tsx` file in `ui/src/components` and save it.

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.

There is one important point about privacy here. 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.

1.4.3.2 MessageEdit Component

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

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`.

There is again an important point here, in this case about how *authorization* is enforced. 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.

1.4.3.3 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.

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've saved the file after copying the code.

You can see we simply follow the formatting of the previous panels and 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.

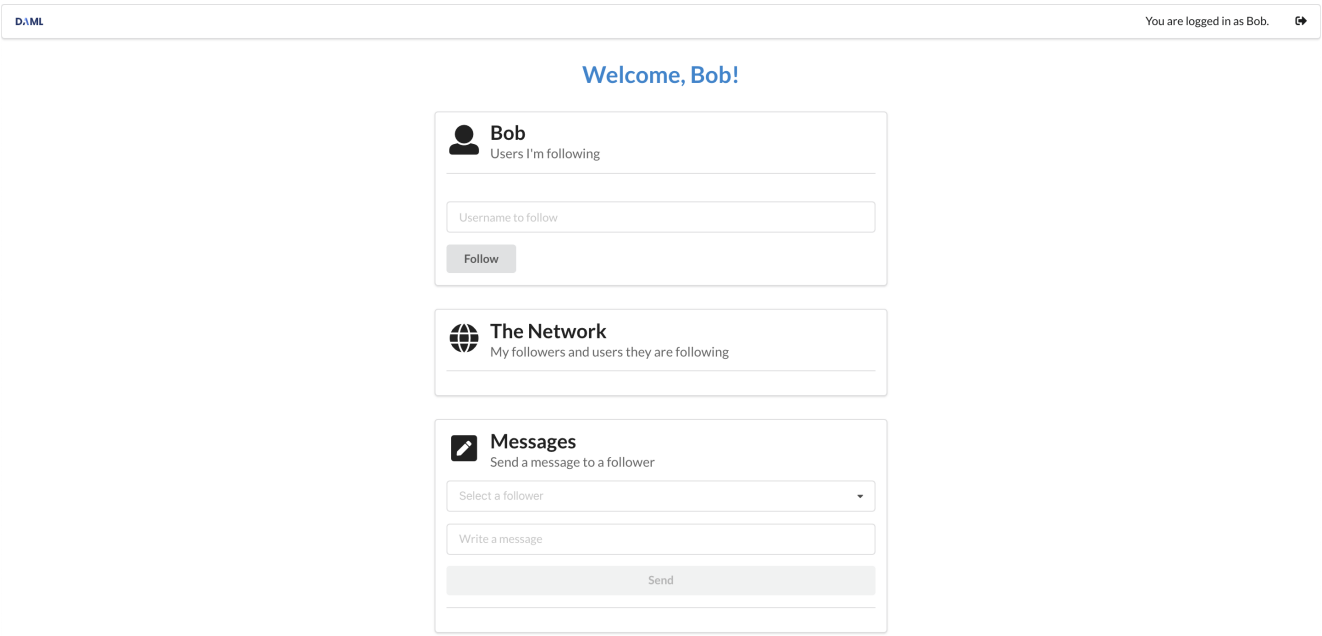
1.4.4 Running the updated UI

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

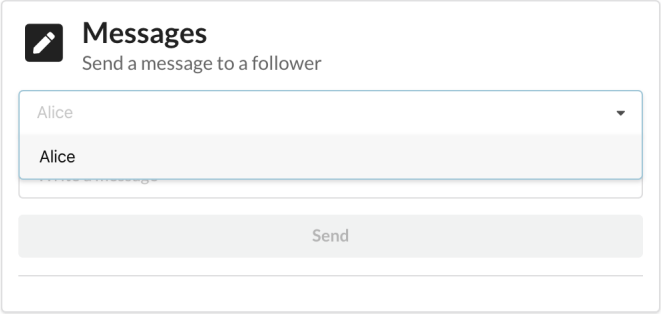
Once you've done all these changes you should see the same login page as before at http://localhost:3000.

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

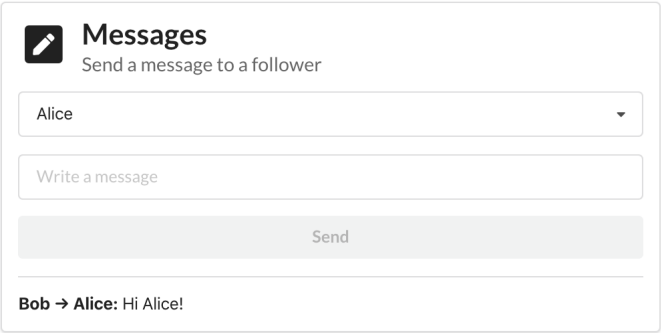
Create DAML App
Username
Log in


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

Go ahead and add follow more users, and log in as some of those users in separate browser windows to follow yourself back. Then, if you click on the dropdown menu in the *Messages* panel, you'll be able to see some followers to message!



Send some messages between users and make sure you can see each one from the other side. You'll notice that new messages appear in the UI as soon as they are 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

1.4.5 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*.

Chapter 2

Writing Daml

2.1 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 Connect'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 Connect SDK*

Next: *1 Basic contracts*.

2.1.1 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`

2.1.1.1 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*.

2.1.1.2 Daml files and modules

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

```
module Token where
```

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

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

2.1.1.3 Templates

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

Listing 1: A simple template

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

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

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

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

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

2.1.1.4 Signatories

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

least one signatory.

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

2.1.1.5 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.

2.1.2 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`

2.1.2.1 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.

2.1.2.2 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 columns, labelled vertically, show which parties know about which contracts. In this simple script, the sole party Alice knows about the contract she created.
The second column shows the ID of the contract. This will be explained later.
The third column shows the status of the contract, either `active` or `archived`.
The remaining 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.

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.

2.1.2.3 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:

```
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'
```

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.

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.

2.1.2.4 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`.

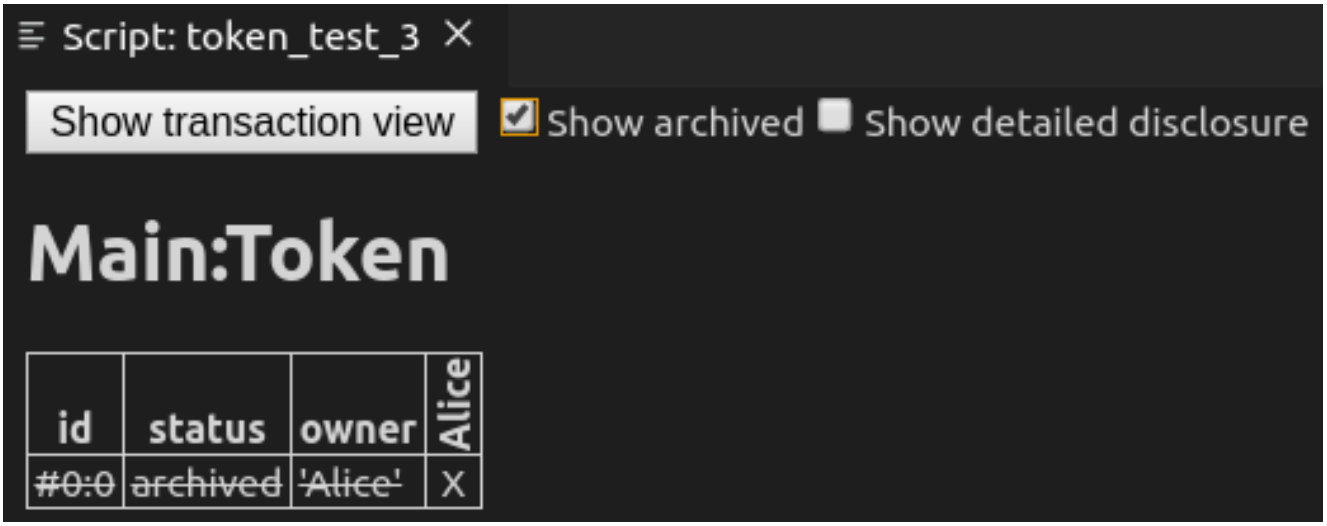
To `archiveCmd` the Token Alice has created, you need to get a handle on its contract ID. In scripts, you do this using `<-` notation. That's because the contract ID needs to be retrieved from the ledger. How this works is discussed in *5 Adding constraints to a contract*.

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"

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

  submitMustFail bob do
    archiveCmd alice_token

  submit alice do
    archiveCmd alice_token
```

2.1.2.5 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.
> `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.

2.1.2.6 Exercises

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

1. Write a template for a second type of Token.
2. Write a script with two parties and two types of tokens, creating one token of each type for each party and archiving one token for each party, leaving one token of each type in the final ledger view.
3. In *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.

```
Script: token_test_3 ×
Show table view

Transactions:
  TX 0 1970-01-01T00:00:00Z (Main:106:18)
  #0:0
  │   consumed by: #1:0
  │   referenced by #1:0
  │   known to (since): 'Alice' (0)
  └─> create Main:Token
      with
        owner = 'Alice'

  TX 1 1970-01-01T00:00:00Z (Main:112:3)
  #1:0
  │   known to (since): 'Alice' (1)
  └─> 'Alice' exercises Archive on #0:0 (Main:Token)
              with

Active contracts:

Return value: {}
```

2.1.2.7 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.

2.1.3 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`

2.1.3.1 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"
```

2.1.3.2 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)
```

(continues on next page)

(continued from previous page)

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

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

You can access the data in the tuples using:

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

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

Lists

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

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

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

```
import DA.List
import Daml.Script

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

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

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

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

(continues on next page)

(continued from previous page)

```
  -- `::` 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"
      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
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
    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
  accountant <- allocateParty "Bob"
```

(continues on next page)

(continued from previous page)

```
  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
```

(continues on next page)

(continued from previous page)

```
  | InAccount Account
    deriving (Eq, Show)

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

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

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

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

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

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

```
data Optional a
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
    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
      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
```

2.1.3.3 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
```

(continues on next page)

(continued from previous page)

```
    -- 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.

2.1.3.4 Contract keys

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

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

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

template Account
  with
    accountant : Party
    owner : Party
    number : Text
```

(continues on next page)

(continued from previous page)

```
    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
      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
```

(continues on next page)

(continued from previous page)

```
      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
  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
```

(continues on next page)

(continued from previous page)

```
    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
    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
```

(continues on next page)

(continued from previous page)

```
        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.

2.1.3.5 Next up

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

In *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.

2.1.4 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`

2.1.4.1 Choices as methods

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

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

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

    controller owner can
      UpdateTelephone
        : ContractId Contact
        with
          newTelephone : Text
        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, `controller owner can` says that the following choices are *controlled* by `owner`, meaning `owner` is the only party that is allowed to *exercise* them. The line starts a new block in which multiple choices can be defined.
> `UpdateTelephone` is the name of a 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 `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

  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.

2.1.4.2 Choices as delegation

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

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

```
  controller party can
    UpdateAddress
      : ContractId Contact
      with
        newAddress : Text
      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`. Controllers specified via `controller c can` syntax become *observers* of the contract. More on *observers* later, but in short, they get to see any changes to the contract.

2.1.4.3 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

`nonconsuming`. `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)

  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:
```

(continues on next page)

(continued from previous page)

```
      #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.

2.1.4.4 A simple cash model

With the power of choices, you can build your first interesting model: issuance of cash IOUs (I owe you). The model presented here is simpler than the one in *3 Data types* as it's not concerned with the location of the physical cash, but merely with liabilities:

```
-- Copyright (c) 2021 Digital Asset (Switzerland) GmbH and/or its
↪affiliates. All rights reserved.
-- SPDX-License-Identifier: Apache-2.0


module SimpleIou where

import Daml.Script

data Cash = Cash with
  currency : Text
  amount : Decimal
```

(continues on next page)

(continued from previous page)

```
    deriving (Eq, Show)

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

    controller owner can
      Transfer
        : ContractId SimpleIou
        with
          newOwner : Party
        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.

2.1.4.5 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.

2.1.5 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`

2.1.5.1 Template preconditions

The first kind of restriction you may want to put on the contract model are called *template preconditions*. 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

    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"
```

2.1.5.2 Assertions

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

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

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

```
    controller owner can
      Transfer
        : ContractId SimpleIou
        with
          newOwner : Party
        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.)

```
    controller owner can
      Redeem
        : ()
        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.
  submitMustFail bob do
```

(continues on next page)

(continued from previous page)

```
    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.

2.1.5.3 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
```

2.1.5.4 Actions and `do` blocks

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

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

Pure expressions compared to Actions

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

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

```
now <- getTime
```

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

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

Actions and impurity

Actions are a way to handle such impure expressions. `Action a` is a type class with a single parameter `a`, and `Update` and `Script` are instances of `Action`. A value of such a type `m a` where `m`

is an instance of `Action` can be interpreted as a recipe for an action of type `m`, which, when executed, returns a value `a`.

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

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

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

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

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

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

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`.

2.1.5.5 Failing actions

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

the ledger.

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

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

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

2.1.5.6 A sample Action

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

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

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

flipCoin : CoinGame Face
getCoin : Script Coin
```

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

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

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

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

(continues on next page)

(continued from previous page)

```
      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.

2.1.5.7 Errors

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

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

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

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

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

2.1.5.8 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.

2.1.6 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`

2.1.6.1 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
```

(continues on next page)

(continued from previous page)

```
    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

    controller owner can
      Transfer
        : ContractId Iou
        with
          newOwner : Party
        do
          assertMsg "newOwner cannot be equal to owner." (owner /=
↪newOwner)
```

(continues on next page)

(continued from previous page)

```
          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.

2.1.6.2 Use propose-accept workflows for one-off authorization

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

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

(continues on next page)

(continued from previous page)

```
    controller iou.owner can
      IouProposal_Accept
        : ContractId Iou
        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)

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

    controller newOwner can
      IouTransferProposal_Reject
        : ContractId Iou
        do
          create iou
```

(continues on next page)

(continued from previous page)

```
    IouTransferProposal_Accept
      : ContractId Iou
      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`:

```
    ProposeTransfer
      : ContractId IouTransferProposal
      with
        newOwner : Party
      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.
```

(continues on next page)

(continued from previous page)

```
  tpbc <- submit bob do
    exerciseCmd iou2 ProposeTransfer with
      newOwner = charlie

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

2.1.6.3 Use role contracts for ongoing authorization

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

Jointly, an `owner` and `newOwner` can transfer an asset, as demonstrated in the script above. In *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

    controller sender can
      nonconsuming Send_Iou
        : ContractId Iou
        with
          iouCid : ContractId Iou
```

(continues on next page)

(continued from previous page)

```
      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
```

2.1.6.4 Daml's authorization model

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

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

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

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

The required authorizers of actions are:

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

The authorizers of transactions are:

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

An authorization example

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

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

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

```
TX #12 1970-01-01T00:00:00Z (Parties: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
```

(continues on next page)

(continued from previous page)

```
                              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

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

    controller partyB can
      TryB
        : ContractId NonTransitive
          with
            other : ContractId NonTransitive
        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.

2.1.6.5 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.

2.1.7 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`

2.1.7.1 Daml projects

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

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

Take a look at the `daml.yaml` for the chapter 7 project:

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

(continues on next page)

(continued from previous page)

```
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*.

2.1.7.2 Project structure

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

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

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

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

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

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

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

```
import Intro.Asset
```

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

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

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

```
import Daml.Script
```

2.1.7.3 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.

2.1.7.4 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`

```
    Split
      : SplitResult
      with
        splitQuantity : Decimal
      do
        splitAsset <- create this with
          quantity = splitQuantity
        remainder <- create this with
          quantity = quantity - splitQuantity
        return SplitResult with
          splitAsset
          remainder

    Merge
      : ContractId Asset
      with
        otherCid : ContractId Asset
      do
        other <- fetch otherCid
        assertMsg
          "Merge failed: issuer does not match"
          (issuer == other.issuer)
        assertMsg
```

(continues on next page)

(continued from previous page)

```
            "Merge failed: owner does not match"
            (owner == other.owner)
          assertMsg
            "Merge failed: symbol does not match"
            (symbol == other.symbol)
          archive otherCid
          create this with
            quantity = quantity + other.quantity
```

The `return` function used in `Split` is available in any `Action` context. The result of `return x` is a no-op containing the value `x`. It has an alias `pure`, indicating that it's a pure value, as opposed to a value with side-effects. The `return` name makes sense when it's used as the last statement in a `do` block as its argument is indeed the `return`-value of the `do` block in that case.

Taking transaction composition a step further, the `Trade_Settle` choice on `Trade` composes two `exercise` actions:

```
      Trade_Settle
        : (ContractId Asset, ContractId Asset)
        with
          quoteAssetCid : ContractId Asset
          baseApprovalCid : ContractId TransferApproval
        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
```

(continues on next page)

(continued from previous page)

```
│   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
    │   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)
```

(continues on next page)

(continued from previous page)

```
        #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.

2.1.7.5 Daml's execution model

Daml's execution model is fairly easy to understand, but has some important consequences. You can imagine the life of a transaction as follows:

Command Submission A user submits a list of Commands via the Ledger API of a Participant Node, acting as a *Party* hosted on that Node. That party is called the requester.

Interpretation Each Command corresponds to one or more Actions. During this step, the `Update` corresponding to each Action is evaluated in the context of the ledger to calculate all consequences, including transitive ones (consequences of consequences, etc.). The result of this is a complete Transaction. Together with its requestor, this is also known as a Commit.

Blinding On ledgers with strong privacy, projections (see *Privacy*) for all involved parties are created. This is also called *projecting*.

Transaction Submission The Transaction/Commit is submitted to the network.

Validation The Transaction/Commit is validated by the network. Who exactly validates can differ from implementation to implementation. Validation also involves scheduling and collision detection, ensuring that the transaction has a well-defined place in the (partial) ordering of Commits, and no double spends occur.

Commitment The Commit is actually committed according to the commit or consensus protocol of the Ledger.

Confirmation The network sends confirmations of the commitment back to all involved Participant Nodes.

Completion The user gets back a confirmation through the Ledger API of the submitting Participant Node.

The first important consequence of the above is that all transactions are committed atomically. Either a transaction is committed as a whole and for all participants, or it fails.

That's important in the context of the `Trade_Settle` choice shown above. The choice transfers a `baseAsset` one way and a `quoteAsset` the other way. Thanks to transaction atomicity, there is no chance that either party is left out of pocket.

The second consequence is that the requester of a transaction knows all consequences of their submitted transaction – there are no surprises in Daml. However, it also means that the requester must have all the information to interpret the transaction. We also refer to this as Principle 2 a bit later on this page.

That's also important in the context of `Trade`. In order to allow Bob to interpret a transaction that transfers Alice's cash to Bob, Bob needs to know both about Alice's `Asset` contract, as well as about some way for `Alice` to accept a transfer – remember, accepting a transfer needs the authority of `issuer` in this example.

2.1.7.6 Observers

Observers are Daml's mechanism to disclose contracts to other parties. They are declared just like signatories, but using the `observer` keyword, as shown in the `Asset` template:

```
template Asset
  with
    issuer : Party
    owner : Party
    symbol : Text
    quantity : Decimal
    observers : [Party]
```

(continues on next page)

(continued from previous page)

```
  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 via the `controller cs can` syntax are automatically made observers. Controllers declared in the `choice` syntax are not, as they can only be calculated at the point in time when the choice arguments are known.

2.1.7.7 Privacy

Daml's privacy model is based on two principles:

Principle 1. Parties see those actions that they have a stake in. Principle 2. Every party that sees an action sees its (transitive) consequences.

Principle 2 is necessary to ensure that every party can independently verify the validity of every transaction they see.

A party has a stake in an action if

> they are a required authorizer of it
> they are a signatory of the contract on which the action is performed
> they are an observer on the contract, and the action creates or archives it

What does that mean for the `exercise tradeCid Trade_Settle` action from `test_trade`?

Alice is the signatory of `tradeCid` and Bob a required authorizer of the `Trade_Settled` action, so both of them see it. According to 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.

2.1.7.8 Next up

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

2.1.8 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
    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
```

(continues on next page)

(continued from previous page)

```
            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*.

2.1.8.1 Next up

We have now seen how to develop safe models and how we can handle errors in those models in a robust and simple way. But the journey doesn't stop there. In *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.

2.1.9 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`.

2.1.9.1 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.

2.1.9.2 Hashes and Identifiers

Under the heading DAR archive contains the following packages: you get a similar looking list of package names, paired with only the long random string repeated. That hexadecimal string, `887056cbb313b94ab9a6caf34f7fe4fbfe19cb0c861e50d1594c665567ab7625` in this case, is the package hash and the primary and only identifier for a package that's guaranteed to be available and preserved. Meta information like name (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 "887056cbb313b94ab9a6caf34f7fe4fbfe19cb0c861e50d1594c665 -o assets_ledger.dar`, making sure to replace the hash with the appropriate one.
4. Run `daml damlc inspect-dar assets_ledger.dar`

You'll notice two things. Firstly, a lot of the dependencies have lost their names, they are now only identifiable by hash. We could of course also create a second project `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.

2.1.9.3 Dependencies and Data Dependencies

Dependencies under the `daml.yaml dependencies` group rely on the `*.hi` files. The information in these files is crucial for dependencies like the Standard Library, which provide functions, types and typeclasses.

However, as you can see above, this information isn't preserved. Furthermore, preserving this information may not even be desirable. Imagine we had built `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.

2.1.9.4 Structuring Projects

As you've seen here, identifiers depend on the package as a whole and packages always bring all their dependencies with them. Thus changing anything in a complex dependency graph can have significant repercussions. It is therefore advisable to keep dependency graphs simple, and to separate concerns which are likely to change at different rates into separate packages.

For example, in all our projects in this intro, including this chapter, our scripts are in the same project as our templates. In practice, that means changing a test changes all identifiers, which is not desirable. It's better for maintainability to separate tests from main templates. If we had done that in 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.

2.1.9.5 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.

2.1.10 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.

2.1.10.1 The Haskell Connection

The previous chapters of this introduction to Daml have mostly covered the structure of templates, and their connection to the *Daml Ledger Model*. The logic of what happens within the `do` blocks of choices has been kept relatively simple. In this chapter, we will dive deeper into Daml's expression language, the part that allows you to write logic inside those `do` blocks. But we can only scratch the surface here. Daml borrows a lot of its language from Haskell. If you want to dive deeper, or learn about specific aspects of the language you can refer to standard literature on Haskell. Some recommendations:

Finding Success and Failure in Haskell (Julie Maronuki, Chris Martin)
Haskell Programming from first principles (Christopher Allen, Julie Moronuki)
Learn You a Haskell for Great Good! (Miran Lipova a)
Programming in Haskell (Graham Hutton)
Real World Haskell (Bryan O'Sullivan, Don Stewart, John Goerzen)

When comparing Daml to Haskell it's worth noting:

Haskell is a lazy language, which allows you to write things like `head [1..]`, meaning take the first element of an infinite list . Daml by contrast is strict. Expressions are fully evaluated, which means it is not possible to work with infinite data structures.
Daml has a `with` syntax for records, and dot syntax for record field access, neither of which 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.

2.1.10.2 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:

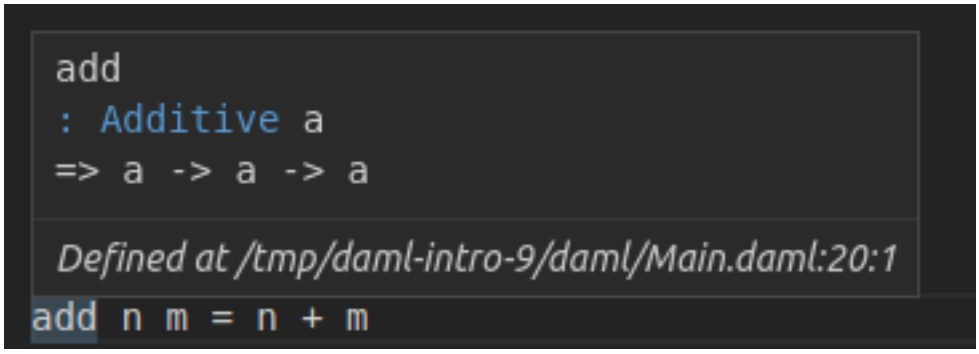
```
increment : Int -> Int
increment n = n + 1
```

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
    (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)`.

2.1.10.3 Control Flow

In this section, we will cover branching and looping, and look at a few common patterns of how to translate procedural code into functional code.

Branching

Until 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
    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
```

(continues on next page)

(continued from previous page)

```
  | 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_`.

2.1.10.4 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.

2.1.11 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.

2.1.11.1 The Prelude

You've already used a lot of functions, types, and typeclasses without importing anything. Functions like `create`, `exercise`, and `(==)`, types like `[]`, `(,)`, `Optional`, and typeclasses like `Eq`, `Show`, and `Ord`. These all come from the *Prelude*. The Prelude is module that gets implicitly imported into every other Daml module and contains both Daml specific machinery as well as the essentials needed to work with the inbuilt types and typeclasses.

2.1.11.2 Important Types from the Prelude

In addition to the *Native types*, the Prelude defines a number of common types:

Lists

You've already met lists. Lists have two constructors `[]` and `x :: xs`, the latter of which is prepend in the sense that `1 :: [2] == [1, 2]`. In fact `[1,2]` is just syntactical sugar for `1 :: 2 :: []`.

Tuples

In addition to the 2-tuple you have already seen, the Prelude contains definitions for tuples of size up to 15. Tuples allow you to store mixed data in an ad-hoc fashion. Common use-cases are return values from functions consisting of several pieces or passing around data in folds, as you saw in *Folds*. An example of a relatively wide Tuple can be found in the test modules of the 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.

2.1.11.3 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.

2.1.11.4 Important Typeclasses from the Prelude

Eq

The `Eq` typeclass allows values of a type to be compared for (in)-equality. It makes available two function: `==` and `/=`. Most data types from the Standard Library have an instance of `Eq`. As you already learned in *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`.

2.1.11.5 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.

2.1.11.6 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.

2.1.11.7 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.

2.1.12 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.

2.1.12.1 Daml Test Tooling

There are three primary tools available in the SDK to test and interact with Daml contracts. It is highly recommended to explore the respective docs. The 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.

2.1.12.2 Debug, Trace, and Stacktraces

The above demonstrates nicely how to test the happy path, but what if a function doesn't behave as you expected? Daml has two functions that allow you to do fine-grained printf debugging: `debug` and `trace`. Both allow you to print something to StdOut if the code is reached. The difference between `debug` and `trace` is similar to the relationship between `abort` and `error`:

`debug : Text -> m ()` maps a text to an Action that has the side-effect of printing to StdOut.
`trace : Text -> a -> a` prints to StdOut when the expression is evaluated.

```
daml> let a : Script () = debug "foo"
daml> let b : Script () = trace "bar" (debug "baz")
[Daml.Script:378]: "bar"
daml> a
[DA.Internal.Prelude:532]: "foo"
daml> b
[DA.Internal.Prelude:532]: "baz"
daml>
```

If in doubt, use `debug`. It's the easier of the two to interpret the results of.

The thing in the square brackets is the last location. It'll tell you the Daml file and line number that triggered the printing, but often no more than that because full stacktraces could violate subtransaction privacy quite easily. If you want to enable stacktraces for some purely functional code in your modules, you can use the machinery in *Module DA.Stack* to do so, but we won't cover that any further here.

2.1.12.3 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.

2.1.12.4 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.

2.2 Language reference docs

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

2.2.1 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*.

2.2.1.1 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.

2.2.1.2 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: 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.

2.2.1.3 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`

2.2.2 Reference: templates

This page gives reference information on templates:

For the structure of a template, see *Overview: template structure*.

2.2.2.1 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).

2.2.2.2 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]`).

2.2.2.3 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.

2.2.2.4 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.

2.2.2.5 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.

2.2.2.6 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: controller first
controller exampleParty can
  NameOfChoice2
      : () -- replace () with the actual return type
    with
      exampleParameter : Text -- parameters here
    do
      return () -- replace this line with the choice body
  nonconsuming NameOfChoice3
      : ()  -- replace () with the actual return type
    with
      exampleParameter : Text -- parameters here
    do
      return () -- replace this line with the choice body
```

A right that the contract gives the controlling party. Can be *exercised*.
This is essentially where all the logic of the template goes.
By default, choices are *consuming*: that is, exercising the choice archives the contract, so no further choices can be exercised on it. You can make a choice non-consuming using the `nonconsuming` keyword.
There are two ways of specifying a choice: start with the `choice` keyword or start with the `controller` keyword.
Starting with `choice` lets you pass in a `Party` to use as a controller. But you must make sure to add that party as an `observer`.
See *Reference: choices* for full reference information.

2.2.2.7 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 `<>`.

2.2.2.8 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.

2.2.2.9 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*.

2.2.3 Reference: choices

This page gives reference information on choices. For information on the high-level structure of a choice, see *Overview: template structure*.

2.2.3.1 `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: controller first
controller exampleParty can
  NameOfAnotherChoice :
      () -- replace () with the actual return type
    with
```

(continues on next page)

(continued from previous page)

```
          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
```

2.2.3.2 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: 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.

2.2.3.3 Controllers

Listing 4: Option 1 for specifying choices: choice name first

```
      controller exampleParty
```

Listing 5: Option 2 for specifying choices: 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.

2.2.3.4 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: 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

2.2.3.5 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: 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

2.2.3.6 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: 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.

2.2.3.7 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.

2.2.3.8 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.

2.2.4 Reference: updates

This page gives reference information on Updates. For the structure around them, see *Overview: template structure*.

2.2.4.1 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*.

2.2.4.2 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.

2.2.4.3 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.

2.2.4.4 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*.

2.2.4.5 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.

2.2.4.6 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.

2.2.4.7 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.

2.2.4.8 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.

2.2.4.9 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).

2.2.4.10 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 scenario results).
You could use `assert False` as an alternative.

2.2.4.11 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:

```
      Transfer : ContractId RestrictedPayout
        with newReceiver : Party
        do
          assert (newReceiver /= blacklisted)
          create RestrictedPayout with receiver = newReceiver; giver;
↪blacklisted; qty
```

2.2.4.12 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:

```
      Complete : ()

        do
          -- bind the ledger effective time to the tchoose variable using
↪getTime
          tchoose <- getTime
```

2.2.4.13 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)
```

2.2.4.14 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
```

2.2.4.15 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.

2.2.5 Reference: data types

This page gives reference information on Daml's data types.

2.2.5.1 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.

2.2.5.2 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 =
  scenario 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.

2.2.5.3 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`.

2.2.5.4 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`.

2.2.5.5 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:

```
optionalIntegerToText (x : Optional Int) : Text =
  case x of
    None -> "Box is empty"
```

(continues on next page)

(continued from previous page)

```
    Some val -> "The content of the box is " <> show val

optionalIntegerToTextTest =
  scenario do
    let
      x = Some 3
    assert (optionalIntegerToText x == "The content of the box is 3")
```

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`:

```
  let
    l = [1, 2, 3]
  in case l of
    [] -> "List is empty"
    _ :: [] -> "List has one element"
    _ :: _ :: _ -> "List has at least two elements"
```

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.

2.2.6 Reference: built-in functions

This page gives reference information on functions for.

2.2.6.1 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.

2.2.6.2 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`.

2.2.6.3 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`

2.2.6.4 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 =
  scenario do
    assert (foldl (+) 0 [1, 2, 3] == 6)
```

2.2.7 Reference: expressions

This page gives reference information for Daml expressions that are not *updates*.

2.2.7.1 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`)

2.2.7.2 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`.

2.2.7.3 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

2.2.7.4 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`.

2.2.7.5 If-then-else

You can use conditional *if-then-else* expressions, for example:

```
if owner == scroogeMcDuck then "sell" else "buy"
```

2.2.7.6 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` and `scenario` blocks:

```
blah = scenario
  do
    let
      x = 1
      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.

    controller owner can
      Transfer : ContractId Iou
        with newOwner : Party
        do
          updateOwner newOwner
```

2.2.8 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*.

2.2.8.1 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
```

2.2.8.2 Partial application

The type of the `tubeSurfaceArea` function described previously, is `Decimal -> Decimal -> Decimal`. An equivalent, but more instructive, way to read its type is: `Decimal -> (Decimal -> Decimal)`: saying that `tubeSurfaceArea` is a function that takes *one* argument and returns another function.

So `tubeSurfaceArea` expects one argument of type `Decimal` and returns a function of type `Decimal -> Decimal`. In other words, this function returns another function. *Only the last application of an argument yields a non-function.*

This is called *currying*: currying is the process of converting a function of multiple arguments to a function that takes just a single argument and returns another function. In Daml, all functions are curried.

This doesn't affect things that much. If you use functions in the classical way (by applying them to all parameters) then there is no difference.

If you only apply a few arguments to the function, this is called *partial application*. The result is a function with partially defined arguments. For example:

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

multiplyTwoNumbersWith7 = multiplyThreeNumbers 7

multiplyWith21 = multiplyTwoNumbersWith7 3

multiplyWith18 = multiplyThreeNumbers 3 6
```

You could also define equivalent lambda functions:

```
multiplyWith18_v2 : Int -> Int
multiplyWith18_v2 xx =
  multiplyThreeNumbers 3 6 xx
```

2.2.8.3 Functions are values

The function type can be explicitly added to the `tubeSurfaceArea` function (when it is written with the lambda notation):

```
-- Type synonym for Decimal -> Decimal -> Decimal
type BinaryDecimalFunction = Decimal -> Decimal -> Decimal

pi : Decimal = 3.1415926535

tubeSurfaceArea : BinaryDecimalFunction =
  \ (r : Decimal) (h : Decimal) -> 2.0 * pi * r * h
```

Note that `tubeSurfaceArea : BinaryDecimalFunction = ...` follows the same pattern as when binding values, e.g., `pi : Decimal = 3.14159265359`.

Functions have types, just like values. Which means they can be used just like normal variables. In fact, in Daml, functions are values.

This means a function can take another function as an argument. For example, define a function `applyFilter: (Int -> Int -> Bool) -> Int -> Int -> Bool` which applies the first argument, a higher-order function, to the second and the third arguments to yield the result.

```
applyFilter (filter : Int -> Int -> Bool)
    (x : Int)
    (y : Int) = filter x y

compute = scenario do
    assert (applyFilter (<) 3 2 == False)
    assert (applyFilter (/=) 3 2 == True)

    assert (round (2.5 : Decimal) == 3)
    assert (round (3.5 : Decimal) == 4)

    assert (explode "me" == ["m", "e"])

    assert (applyFilter (\a b -> a /= b) 3 2 == True)
```

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.

2.2.8.4 Generic functions

A function is *parametrically polymorphic* if it behaves uniformly for all types, in at least one of its type parameters. For example, you can define function composition as follows:

```
compose (f : b -> c) (g : a -> b) (x : a) : c = f (g x)
```

where `a`, `b`, and `c` are any data types. Both `compose ((+) 4) ((*) 2) 3 == 10` and `compose not ((&&) True) False` evaluate to `True`. Note that `((+) 4)` has type `Int -> Int`, whereas `not` has type `Bool -> Bool`.

You can find many other generic functions including this one in the *Daml standard library*.

Note: Daml currently does not support generic functions for a specific set of types, such as `Int` and `Decimal` numbers. For example, `sum (x: a) (y: a) = x + y` is undefined when `a` equals the type `Party`. *Bounded polymorphism* might be added to Daml in a later version.

2.2.9 Reference: scenarios

This page gives reference information on scenario syntax, used for testing templates.

Note that for new project, we recommend Daml Script. For an introduction to Daml Script, see *2 Testing templates using Daml Script*.

2.2.9.1 Scenario keyword

`scenario` function. Introduces a series of transactions to be submitted to the ledger.

2.2.9.2 Submit

`submit` keyword.
Submits an action (a create or an exercise) to the ledger.
Takes two arguments, the party submitting followed by the expression, for example: `submit bankOfEngland do create ...`

2.2.9.3 submitMustFail

`submitMustFail` keyword.
Like `submit`, but you're asserting it should fail.
Takes two arguments, the party submitting followed by the expression by a party, for example: `submitMustFail bankOfEngland do create ...`

2.2.9.4 Scenario body

Updates

Usually *create* and *exercise*. But you can also use other updates, like *assert* and *fetch*.
Parties can only be named explicitly in scenarios.

Passing time

In a scenario, you may want time to pass so you can test something properly. You can do this with `pass`.

Here's an example of passing time:

```
timeTravel =
  scenario do
    -- Get current ledger effective time
    t1 <- getTime
    assert (t1 == datetime 1970 Jan 1 0 0 0)

    -- Pass 1 day
    pass (days 1)
```

(continues on next page)

(continued from previous page)

```
    -- Get new ledger effective time
    t2 <- getTime
    assert (t2 == datetime 1970 Jan 2 0 0 0)
```

Binding variables

As in choices, you can *bind to variables*. Usually, you'd bind commits to variables in order to get the returned value (usually the contract).

2.2.10 Reference: Daml file structure

This page gives reference information on the structure of Daml files outside of *templates*.

2.2.10.1 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`.

2.2.10.2 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.

2.2.10.3 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.

2.2.10.4 Comments

Use `--` for a single line comment. Use `{-` and `-}` for a comment extending over multiple lines.

2.2.10.5 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.

2.2.11 Reference: Daml packages

This page gives reference information on Daml package dependencies.

2.2.11.1 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.

2.2.11.2 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
```

(continues on next page)

(continued from previous page)

```
create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/User.hi
create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/User.hie
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",
```

(continues on next page)

(continued from previous page)

```
            "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",
    "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.

2.2.11.3 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.

2.2.11.4 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`.

2.2.12 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
```

2.2.12.1 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.

2.2.12.2 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.

2.2.12.3 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
```

(continues on next page)

(continued from previous page)

```
  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`.

2.2.12.4 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`.

2.2.12.5 Example

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

```
-- Copyright (c) 2021 Digital Asset (Switzerland) GmbH and/or its
↪affiliates. All rights reserved.
-- SPDX-License-Identifier: Apache-2.0

module Keys where

import DA.Optional

template Keyed
  with
    sig : Party
    obs : Party
  where
    signatory sig
    observer obs

    key sig : Party
    maintainer key

template Divulger
  with
    divulgee : Party
    sig : Party
  where
    signatory divulgee

    controller sig can
      nonconsuming DivulgeKeyed
        : Keyed
```

(continues on next page)

(continued from previous page)

```
        with
          keyedCid : ContractId Keyed
        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
        fetch cid

    nonconsuming choice VisibleKeyed
      : Bool
      with
        key : Party
        delegee : Party
      controller delegee
      do
        visibleByKey @Keyed key
```

(continues on next page)

(continued from previous page)

```
    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

lookupTest = scenario do

  -- Put four parties in the four possible relationships with a `Keyed`
  sig <- getParty "s" -- Signatory
  obs <- getParty "o" -- Observer
  divulgee <- getParty "d" -- Divulgee
  blind <- getParty "b" -- Blind

  keyedCid <- submit sig do create Keyed with ..
  divulgercid <- submit divulgee do create Divulger with ..
  submit sig do exercise divulgercid DivulgeKeyed with ..

  -- Now the signatory and observer delegate their choices
  sigDelegationCid <- submit sig do
    create Delegation with
      sig
      delegees = [obs, divulgee, blind]
  obsDelegationCid <- submit obs do
    create Delegation with
      sig = obs
      delegees = [divulgee, blind]

  -- TESTING LOOKUPS AND FETCHES

  -- Maintainer can fetch
  submit sig do
    (cid, keyed) <- fetchByKey @Keyed sig
    assert (keyedCid == cid)
  -- Maintainer can see
  submit sig do
    b <- visibleByKey @Keyed sig
```

(continues on next page)

(continued from previous page)

```
  assert b
-- Maintainer can lookup
submit sig do
  mcid <- lookupByKey @Keyed sig
  assert (mcid == Some keyedCid)

-- Stakeholder can fetch
submit obs do
  (cid, l) <- fetchByKey @Keyed sig
  assert (keyedCid == cid)
-- Stakeholder can't see without authorization
submitMustFail obs do visibleByKey @Keyed sig
-- Stakeholder can see with authorization
submit obs do
  b <- exercise sigDelegationCid VisibleKeyed with
    delegee = obs
    key = sig
  assert b
-- Stakeholder can't lookup without authorization
submitMustFail obs do lookupByKey @Keyed sig
-- Stakeholder can lookup with authorization
submit obs do
  mcid <- exercise sigDelegationCid LookupKeyed with
    delegee = obs
    lookupKey = sig
  assert (mcid == Some keyedCid)

-- Divulgee _can_ fetch the contract directly
submit divulgee do
  exercise obsDelegationCid UnkeyedFetch with
      delegee = divulgee
      cid = keyedCid
-- Divulgee can't fetch through the key
submitMustFail divulgee do fetchByKey @Keyed sig
-- Divulgee can't see
submitMustFail divulgee do visibleByKey @Keyed sig
-- Divulgee can't see with stakeholder authority
submitMustFail divulgee do
  exercise obsDelegationCid VisibleKeyed with
      delegee = divulgee
      key = sig
-- Divulgee can't lookup
submitMustFail divulgee do lookupByKey @Keyed sig
-- Divulgee can't lookup with stakeholder authority
submitMustFail divulgee do
  exercise obsDelegationCid LookupKeyed with
      delegee = divulgee
      lookupKey = sig
-- Divulgee can't do positive lookup with maintainer authority.
```

(continues on next page)

(continued from previous page)

```
submitMustFail divulgee do
  b <- exercise sigDelegationCid VisibleKeyed with
    delegee = divulgee
    key = sig
  assert $ not b
-- 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
  mcid <- exercise sigDelegationCid LookupKeyed with
    delegee = divulgee
    lookupKey = sig
  assert (isNone mcid)
-- Divulgee can't fetch with stakeholder authority
submitMustFail divulgee do
  (cid, keyed) <- exercise obsDelegationCid FetchKeyed with
    delegee = divulgee
    lookupKey = sig
  assert (keyedCid == cid)

-- Blind party can't fetch
submitMustFail blind do fetchByKey @Keyed sig
-- Blind party can't see
submitMustFail blind do visibleByKey @Keyed sig
-- Blind party can't see with stakeholder authority
submitMustFail blind do
  exercise obsDelegationCid VisibleKeyed with
    delegee = blind
    key = sig
-- Blind party can't see with maintainer authority
submitMustFail blind do
  b <- exercise sigDelegationCid VisibleKeyed with
    delegee = blind
    key = sig
  assert $ not b
-- Blind party can't lookup
submitMustFail blind do lookupByKey @Keyed sig
-- Blind party can't lookup with stakeholder authority
submitMustFail blind do
  exercise 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
```

(continues on next page)

(continued from previous page)

```
    mcid <- exercise sigDelegationCid LookupKeyed with
      delegee = blind
      lookupKey = sig
    assert (isNone mcid)
  -- Blind party can't fetch with stakeholder authority as lookup is
↪negative
  submitMustFail blind do
    exercise obsDelegationCid FetchKeyed with
      delegee = blind
      lookupKey = sig
  -- Blind party can see nonexistence of a contract
  submit blind do
    b <- exercise obsDelegationCid VisibleKeyed with
      delegee = blind
      key = obs
    assert $ not b
  -- Blind can do a negative lookup on a truly nonexistant contract
  submit blind do
    mcid <- exercise obsDelegationCid LookupKeyed with
      delegee = blind
      lookupKey = obs
    assert (isNone mcid)
  -- TESTING CREATES AND ARCHIVES

  -- Divulgee can archive
  submit divulgee do
    exercise sigDelegationCid ArchiveKeyed with
      delegee = divulgee
      keyedCid
  -- Divulgee can create
  keyedCid2 <- submit divulgee do
    exercise sigDelegationCid CreateKeyed with
      delegee = divulgee
      obs

  -- Stakeholder can archive
  submit obs do
    exercise sigDelegationCid ArchiveKeyed with
      delegee = obs
      keyedCid = keyedCid2
  -- Stakeholder can create
  keyedCid3 <- submit obs do
    exercise sigDelegationCid CreateKeyed with
      delegee = obs
      obs

  return ()
```

2.2.13 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:

2.2.13.1 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.

2.2.13.2 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 <> ")"
```

2.2.13.3 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.

2.2.13.4 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"
```

2.3 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:

2.3.1 Module Prelude

The pieces that make up the Daml language.

2.3.1.1 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 *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 *MapKey* k => *Semigroup* (*Map* k v)

instance *MapKey* a => *Semigroup* (*Set* 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 *MapKey* k => *Monoid* (*Map* k v)

instance *MapKey* a => *Monoid* (*Set* 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 *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 *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* (*Set 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)

class *DamlInterface* **where**

instance *DamlInterface*

2.3.1.2 Data Types

data *AnyChoice*

Existential choice type that can wrap an arbitrary choice.

AnyChoice

Field	Type	Description
getAnyChoice	Any	
getAnyChoiceTemplateTypeRep	*TemplateTypeRep*	

instance *Eq AnyChoice*

instance *Ord AnyChoice*

data *AnyContractKey*

Existential contract key type that can wrap an arbitrary contract key.

AnyContractKey

Field	Type	Description
getAnyContractKey	Any	
getAnyContractKeyTemplateTypeRep	*TemplateTypeRep*	

instance *Eq AnyContractKey*

instance *Ord AnyContractKey*

data *AnyTemplate*

Existential template type that can wrap an arbitrary template.

AnyTemplate

Field	Type	Description
getAnyTemplate	Any	

instance *Eq AnyTemplate*

instance *Ord AnyTemplate*

data *TemplateTypeRep*

Unique textual representation of a template Id.

TemplateTypeRep

Field	Type	Description
getTemplateType-Rep	TypeRep	

instance *Eq TemplateTypeRep*

instance *Ord TemplateTypeRep*

data *Down* a

The `Down` type can be used for reversing sorting order. For example, `sortOn (\x -> Down x.field)` would sort by descending `field`.

Down a

instance *Action Down*

instance *Applicative Down*

instance *Functor Down*

instance *Eq* a => *Eq* (*Down* a)

instance *Ord* a => *Ord* (*Down* a)

instance *Show* a => *Show* (*Down* a)

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* (*Set Party*)

instance *IsParties* (*NonEmpty Party*)

instance *IsParties* (*Set Party*)

instance *IsParties* [*Party*]

instance *MapKey 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** = (*Template* 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.

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]`.

([])

(:) _ _

2.3.1.3 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).

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.

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 : (*HasFetchByKey* t k, *HasExercise* t 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.

2.3.2 Module DA.Action

Action

2.3.2.1 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.

2.3.3 Module DA.Action.State

DA.Action.State

2.3.3.1 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)

2.3.3.2 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.

2.3.4 Module DA.Action.State.Class

DA.Action.State.Class

2.3.4.1 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)

2.3.5 Module DA.Assert

2.3.5.1 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.

2.3.6 Module DA.Bifunctor

2.3.6.1 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)
```

(continues on next page)

(continued from previous page)

```
>>> 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)

2.3.7 Module DA.BigNumeric

This module exposes operations for working with the `BigNumeric` type.

2.3.7.1 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
```

2.3.8 Module DA.Date

2.3.8.1 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*

2.3.8.2 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.

2.3.9 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).

2.3.9.1 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

2.3.10 Module DA.Exception

Exception handling in Daml.

2.3.10.1 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*

2.3.10.2 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*	

2.3.11 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
```

2.3.11.1 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* (*Map* k)

instance *Foldable NonEmpty*

instance *Foldable Set*

instance *Foldable* (*Either* a)

instance *Foldable ([])*

instance *Foldable* a

2.3.11.2 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'.

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.

2.3.12 Module DA.Functor

The `Functor` class is used for types that can be mapped over.

2.3.12.1 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 `()`.

2.3.13 Module DA.List

List

2.3.13.1 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.

2.3.14 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.

2.3.14.1 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
```

2.3.15 Module DA.List.Total

2.3.15.1 Functions

head : *ActionFail* m => [a] -> m a

tail : *ActionFail* m => [a] -> m [a]

last : *ActionFail* m => [a] -> m a

init : *ActionFail* m => [a] -> m [a]

(!!) : *ActionFail* m => [a] -> *Int* -> m a

foldl1 : *ActionFail* m => (a -> a -> a) -> [a] -> m a

foldr1 : *ActionFail* m => (a -> a -> a) -> [a] -> m a

foldBalanced1 : *ActionFail* m => (a -> a -> a) -> [a] -> m a

minimumBy : *ActionFail* m => (a -> a -> *Ordering*) -> [a] -> m 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`. The result is wrapped in a monadic context, with a failure if `xs` is empty.

maximumBy : *ActionFail* m => (a -> a -> *Ordering*) -> [a] -> m 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`. The result is wrapped in a monadic context, with a failure if `xs` is empty.

minimumOn : (*ActionFail* m, *Ord* k) => (a -> k) -> [a] -> m 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`. The result is wrapped in a monadic context, with a failure if `xs` is empty.

maximumOn : (*ActionFail* m, *Ord* k) => (a -> k) -> [a] -> m 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`. The result is wrapped in a monadic context, with a failure if `xs` is empty.

2.3.16 Module DA.Logic

Logic - Propositional calculus.

2.3.16.1 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)

2.3.16.2 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.

2.3.17 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.

2.3.17.1 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.

2.3.18 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.

2.3.18.1 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

2.3.19 Module DA.Monoid

2.3.19.1 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)

2.3.20 Module DA.Next.Map

`DA.Next.Map` is deprecated. Please use `DA.Map` instead.

2.3.20.1 Typeclasses

class *Eq* k => *MapKey* k **where**

A class for types that can be used as keys for the `Map` type. All keys `k` must satisfy `keyFromText (keyToText k) == k`.

keyToText : k -> *Text*

Turn a key into its textual representation. This function must be injective.

keyFromText : *Text* -> k
Recover a key from its textual representation. `keyFromText x` is allowed to fail whenever there is *no* key `k` with `keyToText k == x`. Whenever such a `k` does exist, then it must satisfy `keyFromText x == k`.

instance *MapKey Party*

instance *MapKey Decimal*

instance *MapKey Int*

instance *MapKey Text*

2.3.20.2 Data Types

data *Map* k v

A `Map k v` is an associative array data type composed of a collection of key/value pairs of key type `k` and value type `v` such that each possible key appears at most once in the collection.

instance *Foldable* (*Map* k)

instance *MapKey* k => *Monoid* (*Map* k v)

instance *MapKey* k => *Semigroup* (*Map* k v)

instance *MapKey* k => *Traversable* (*Map* k)

instance *MapKey* k => *Functor* (*Map* k)

instance *Eq* v => *Eq* (*Map* k v)

instance *Ord* v => *Ord* (*Map* k v)

instance (*MapKey* k, *Show* k, *Show* v) => *Show* (*Map* k v)

2.3.20.3 Functions

fromList : *MapKey* k => [(k, v)] -> *Map* k v
Create a map from a list of key/value pairs.

fromListWith : *MapKey* 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 (<>) [(5,"a"), (5,"b"), (3,"b"), (3,"a"), (5,"c")] ==
↪fromList [(3, "ba"), (5, "abc")]
fromListWith (<>) [] == (empty : Map Int Text)
```

toList : *MapKey* k => *Map* k v -> [(k, v)]
Convert the map to a list of key/value pairs where the keys are in ascending order of their textual representation.

fromTextMap : *TextMap* v -> *Map Text* v
Create a `Map` from a `TextMap`.

toTextMap : *MapKey* k => *Map* k v -> *TextMap* v
Convert a `Map` into a `TextMap`.

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 : *MapKey* k => k -> *Map* k v -> *Optional* v
Lookup the value at a key in the map.

member : *MapKey* k => k -> *Map* k v -> *Bool*
Is the key a member of the map?

filter : *MapKey* 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 : *MapKey* 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 : *MapKey* 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 : *MapKey* 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.

union : *MapKey* 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 : *MapKey* k => (k -> a -> *Optional* c) -> (k -> b -> *Optional* c) -> (k -> a -> b -> *Optional* c) -> *Map* k a -> *Map* k b -> *Map* k 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.

2.3.21 Module DA.Next.Set

`DA.Next.Set` is deprecated. Please use `DA.Set` instead.

2.3.21.1 Data Types

data *Set* a

The type of a set.

instance *MapKey* a => *Monoid* (*Set* a)

instance *MapKey* a => *Semigroup* (*Set* a)

instance *IsParties* (*Set Party*)

instance *Eq* (*Set* a)

instance *Ord* (*Set* a)

instance (*MapKey* a, *Show* a) => *Show* (*Set* a)

2.3.21.2 Functions

empty : *Set* a
The empty set.

size : *Set* a -> *Int*
The number of elements in the set.

toList : *MapKey* a => *Set* a -> [a]
Convert the set to a list of elements.

fromList : *MapKey* a => [a] -> *Set* a
Create a set from a list of elements.

toTextMap : *Set Text* -> *TextMap* ()
Convert a `Set` into a `TextMap`.

fromTextMap : *TextMap* () -> *Set Text*
Create a `Set` from a `TextMap`.

member : *MapKey* a => a -> *Set* a -> *Bool*
Is the element in the set?

null : *Set* a -> *Bool*
Is this the empty set?

insert : *MapKey* a => a -> *Set* a -> *Set* a
Insert an element in a set. If the set already contains an element equal to the given value, it is replaced with the new value.

filter : *MapKey* a => (a -> *Bool*) -> *Set* a -> *Set* a
Filter all elements that satisfy the predicate.

delete : *MapKey* a => a -> *Set* a -> *Set* a
Delete an element from a set.

singleton : *MapKey* a => a -> *Set* a
Create a singleton set.

union : *MapKey* a => *Set* a -> *Set* a -> *Set* a
The union of two sets, preferring the first set when equal elements are encountered.

intersection : *MapKey* a => *Set* a -> *Set* a -> *Set* a
The intersection of two sets. Elements of the result come from the first set.

difference : *MapKey* a => *Set* a -> *Set* a -> *Set* a
`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]

2.3.22 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
```

2.3.22.1 Functions

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.

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.

2.3.23 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.

2.3.23.1 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)

2.3.24 Module DA.Numeric

2.3.24.1 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.

2.3.25 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.

2.3.25.1 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.

2.3.26 Module DA.Optional.Total

2.3.26.1 Functions

fromSome : *ActionFail* m => *Optional* a -> m a

fromSomeNote : *ActionFail* m => *Text* -> *Optional* a -> m a

2.3.27 Module DA.Record

Exports the record machinery necessary to allow one to annotate code that is polymorphic in the underlying record type.

2.3.27.1 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

2.3.28 Module DA.Semigroup

2.3.28.1 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)

2.3.29 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.

2.3.29.1 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)

2.3.29.2 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?

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]

2.3.30 Module DA.Stack

2.3.30.1 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.

2.3.30.2 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`.

2.3.31 Module DA.Text

Functions for working with Text.

2.3.31.1 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.

2.3.32 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.

2.3.32.1 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.

2.3.33 Module DA.Time

2.3.33.1 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*

2.3.33.2 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

2.3.34 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
```

2.3.34.1 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 *MapKey* k => *Traversable* (*Map* k)

instance *Traversable NonEmpty*

instance *Traversable* (*Either* a)

instance *Traversable ([])*

instance *Traversable* a

2.3.34.2 Functions

forA : (*Traversable* t, *Applicative* f) => t a -> (a -> f b) -> f (t b)
`forA` is `mapA` with its arguments flipped.

2.3.35 Module DA.Tuple

Tuple - Ubiquitous functions of tuples.

2.3.35.1 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.

2.3.36 Module DA.Validation

`Validation` type and associated functions.

2.3.36.1 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)

2.3.36.2 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`.

2.4 Troubleshooting

2.4.1 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.

2.4.2 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
    controller (supervisor receiver) can
      SupervisedPayout_Call
        returning ContractId Payout
        to create Payout with giver; receiver; amount
```

Hovering over the compilation error displays:

```
[Type checker] Argument expands to non-serializable type Party -> Party.
```

2.4.3 Modeling questions

2.4.3.1 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.

2.4.3.2 How to model rights

Use a contract choice to model a right. A party exercises that right by exercising the choice.

2.4.3.3 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.

2.4.3.4 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.

2.4.3.5 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.

2.4.3.6 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.

2.4.3.7 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.

2.4.4 Testing questions

2.4.4.1 How to test that a contract is visible to a party

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:

```
submit alice do
  result <- fetch invoiceCid
```

You could also check (in the `submit` block) that the contract has some expected values:

```
assert (result == (Invoice with
  payee = alice
  payer = acme
  amount = 130.0
  service = "A job well done"
  timeLimit = datetime 1970 Feb 20 0 0 0))
```

using an equality test and an `assert`:

```
submit alice do
  result <- fetch invoiceCid
  assert (result == (Invoice with
    payee = alice
    payer = acme
    amount = 130.0
    service = "A job well done"
    timeLimit = datetime 1970 Feb 20 0 0 0))
```

2.4.4.2 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.

2.5 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.

2.5.1 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.

2.5.1.1 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.

2.5.1.2 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

    controller issuer can
      nonconsuming Invite : ContractId CoinIssueProposal
        with owner: Party
        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

    controller coinAgreement.owner can
      AcceptCoinProposal
        : ContractId CoinIssueAgreement
        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

    controller issuer can
      nonconsuming Issue : ContractId Coin
        with amount: Decimal
        do create Coin with issuer; owner; amount; delegates = []
```

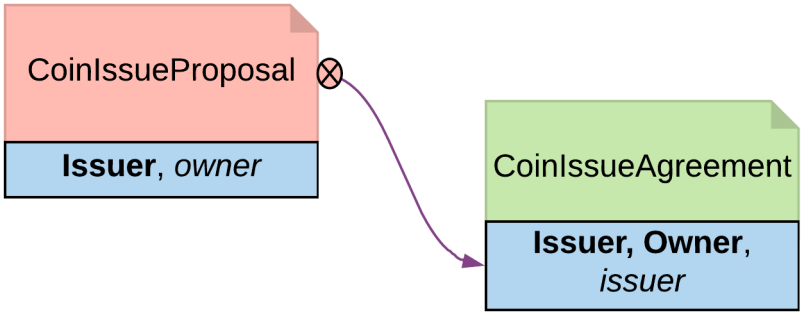


Fig. 1: Initiate and Accept pattern diagram

2.5.1.3 Trade-offs

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

2.5.2 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.

2.5.2.1 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.

2.5.2.2 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]
```

(continues on next page)

(continued from previous page)

```
  where
    signatory signatories
    ensure
      unique signatories
  -- The rest of the template to be agreed to would follow here
```

Pending contract The *Pending* contract needs to contain the contents of the proposed *Agreement* contract, as a parameter. This is so that parties know what they are agreeing to, and also so that when all parties have signed, the *Agreement* contract can be created.
The *Pending* contract has a list of parties who have signed it, and a list of parties who have yet to sign it. If you add these lists together, it has to be the same set of parties as the `signatories` of the *Agreement* contract.
All of the `toSign` parties have the choice to `Sign`. This choice checks that the party is indeed a member of `toSign`, then creates a new instance of the *Pending* contract where they have been moved to the `signed` list.

```
template Pending
  with
    finalContract: Agreement
    alreadySigned: [Party]
  where
    signatory alreadySigned
    observer finalContract.signatories
    ensure
      -- Can't have duplicate signatories
      unique alreadySigned

    -- The parties who need to sign is the finalContract.signatories
↪with alreadySigned filtered out
    let toSign = filter (`notElem` alreadySigned) finalContract.
↪signatories

    choice Sign : ContractId Pending with
        signer : Party
      controller signer
        do
          -- Check the controller is in the toSign list, and if they
↪are, sign the Pending contract
          assert (signer `elem` toSign)
          create this with alreadySigned = signer :: alreadySigned
```

Once all of the parties have signed, any of them can create the final Agreement contract using the `Finalize` choice. This checks that all of the signatories for the *Agreement* have signed the *Pending* contract.

```
    choice Finalize : ContractId Agreement with
        signer : Party
      controller signer
        do
          -- Check that all the required signatories have signed
↪Pending
```

(continues on next page)

(continued from previous page)

```
          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
    create 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
    create 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
    exercise pending Sign with signer = person2
  pending <- person3 `submit` do
    exercise pending Sign with signer = person3
  pending <- person4 `submit` do
    exercise pending Sign with signer = person4

  -- A party can't sign the Pending contract twice
  pendingFailTest <- person3 `submitMustFail` do
    exercise pending Sign with signer = person3
  -- A party can't sign on behalf of someone else
  pendingFailTest <- person3 `submitMustFail` do
    exercise 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
    exercise pending Finalize with signer = person1
```

2.5.3 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.

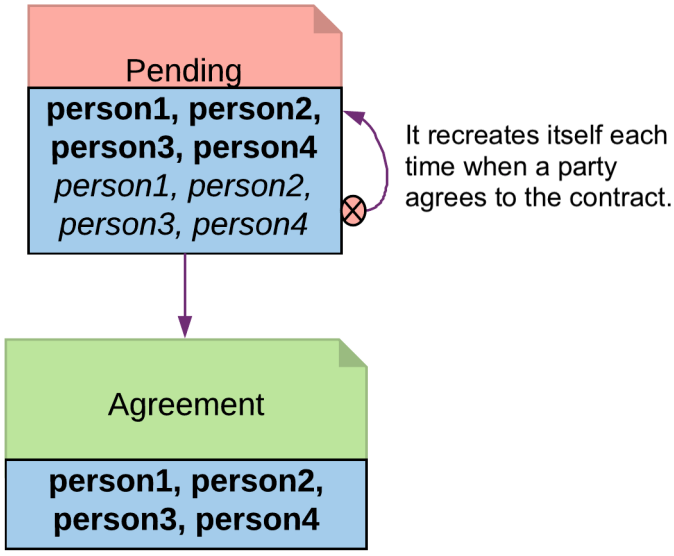


Fig. 2: Multiple Party Agreement Diagram

2.5.3.1 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.

2.5.3.2 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.

```
--the original contract
template Coin
  with
    owner: Party
    issuer: Party
    amount: Decimal
    delegates : [Party]
  where
    signatory issuer, owner
    observer delegates

    controller owner can
```

(continues on next page)

(continued from previous page)

```
      Transfer : ContractId TransferProposal
        with newOwner: Party
        do
            create TransferProposal
             with coin=this; newOwner

      Lock : ContractId LockedCoin
        with maturity: Time; locker: Party
        do create LockedCoin with coin=this; maturity; locker

      Disclose : ContractId Coin
        with p : Party
        do create this with delegates = p :: 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.
    controller issuer can
      Archives
        : ()
        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

    controller principal can
      WithdrawPoA
        : ()
        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.

```
    controller attorney can
      nonconsuming TransferCoin
        : ContractId TransferProposal
        with
          coinId: ContractId Coin
          newOwner: Party
```

(continues on next page)

(continued from previous page)

```
        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:

```
      Disclose : ContractId Coin
        with p : Party
        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.

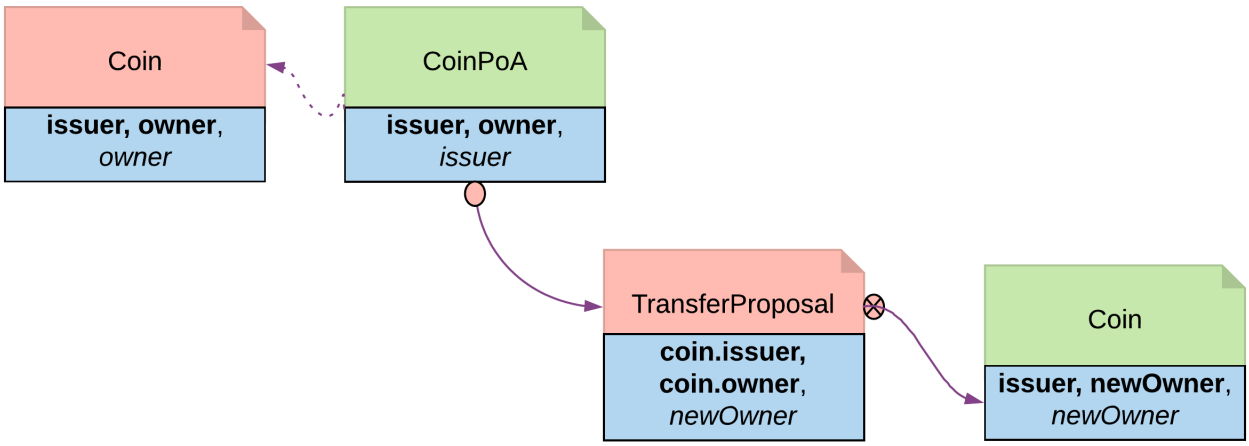


Fig. 3: Delegation pattern diagram

2.5.4 Authorization

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

2.5.4.1 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.

2.5.4.2 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

    controller owner can

      Transfer : ContractId TransferProposal
        with newOwner: Party
        do
            create TransferProposal
             with coin=this; newOwner

      Lock : ContractId LockedCoin
        with maturity: Time; locker: Party
        do create LockedCoin with coin=this; maturity; locker

      Disclose : ContractId Coin
        with p : Party
        do create this with delegates = p :: 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.
    controller issuer can
      Archives
        : ()
        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
```

(continues on next page)

(continued from previous page)

```
    controller issuer can
      WithdrawAuthorization
        : ()
        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.

```
    controller newOwner can
      AcceptTransfer
        : ContractId Coin
        with token: ContractId CoinOwnerAuthorization
        do
          t <- fetch token
          assert (coin.issuer == t.issuer)
          assert (newOwner == t.owner)
          create coin with owner = newOwner
```

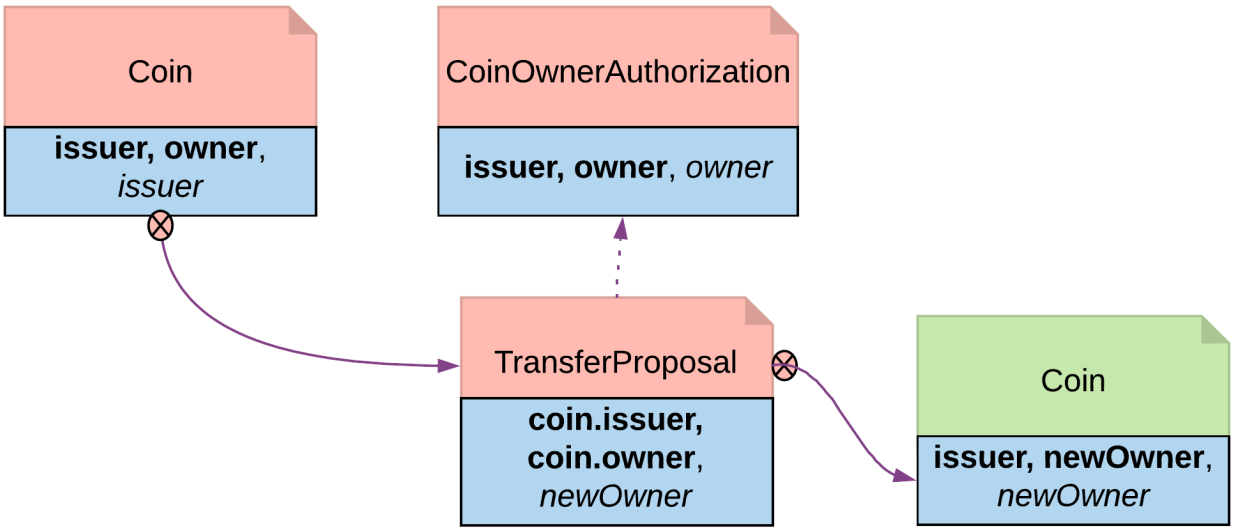


Fig. 4: Authorization Diagram

2.5.5 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.

2.5.5.1 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.

2.5.5.2 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

    controller owner can

      Transfer : ContractId TransferProposal
        with newOwner: Party
        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.
    controller issuer can
      Archives
        : ()
        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.

```
  Lock : ContractId LockedCoin
    with maturity: Time; locker: Party
    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

    controller locker can
      Unlock
        : ContractId Coin
        do create coin
```

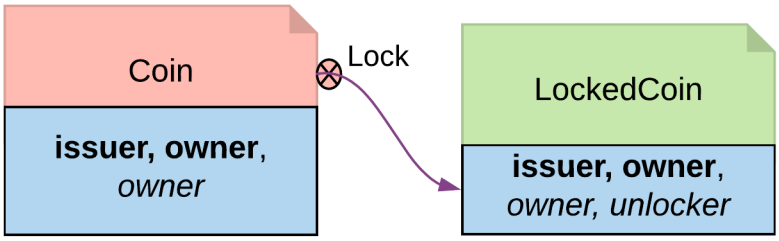


Fig. 5: Locking By Consuming Choice Diagram

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.

```
    controller issuer can
      Archives
        : ()
        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
```

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.

```
    controller owner can
      nonconsuming LockCoin
        : ContractId LockedCoin
        with coinCid: ContractId Coin
             maturity: Time
             locker: Party
        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
```

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.

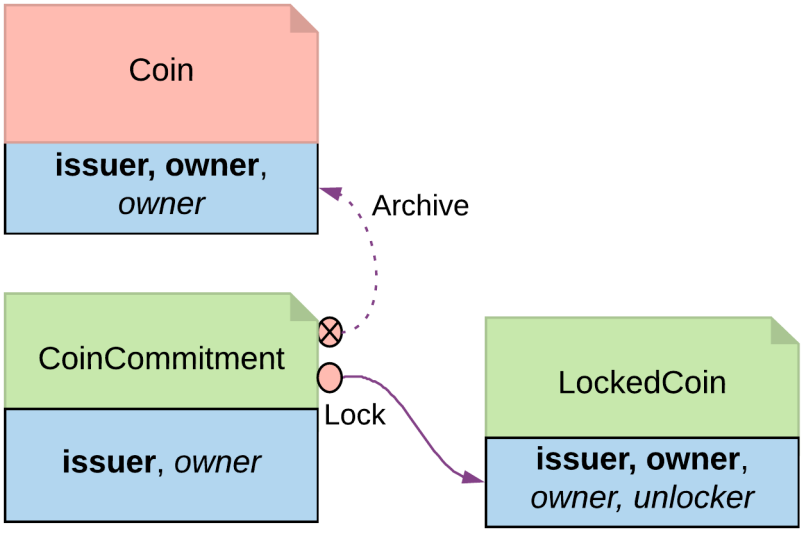


Fig. 6: Locking By Archiving Contract Diagram

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

    controller owner can

      Transfer : ContractId TransferProposal
        with newOwner: Party
        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.
    controller issuer can
      Archives
        : ()
        do assert (issuer == owner)
```

In its original form, all choices are actionable as long as the contract is active. Locking by State

requires introducing fields to track state. This allows for the creation of an active contract in two possible states: locked or unlocked. A Daml modeler can selectively make certain choices actionable only if the contract is in unlocked state. This effectively makes the asset lockable.

The state can be stored in many ways. This example demonstrates how to create a *LockableCoin* through a party. Alternatively, you can add a lock contract to the asset contract, use a boolean flag or include lock activation and expiry terms as part of the template parameters.

Here are the changes we made to the original *Coin* contract to make it lockable.

Add a *locker* party to the template parameters.
Define the states.
- if owner == locker, the coin is unlocked
- if owner != locker, the coin is in a locked state

The contract state is checked on choices.
- *Transfer* choice is only actionable if the coin is unlocked
- *Lock* choice is only actionable if the coin is unlocked and a 3rd party locker is supplied
- *Unlock* is available to the locker party only if the coin is locked

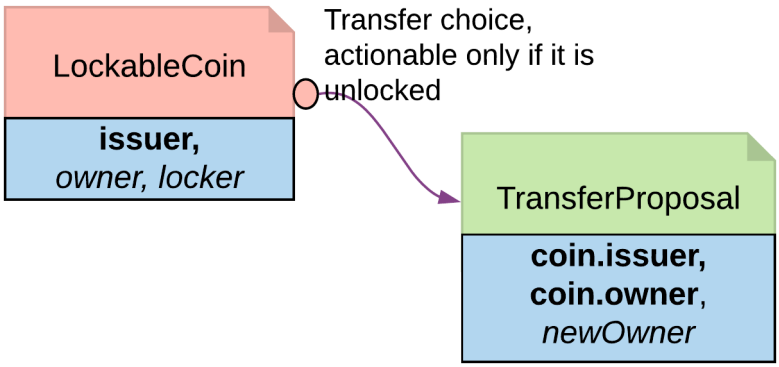
```
template LockableCoin
  with
    owner: Party
    issuer: Party
    amount: Decimal
    locker: Party
  where
    signatory issuer
    signatory owner

    ensure amount > 0.0

    --Transfer can happen only if it is not locked
    controller owner can
      Transfer : ContractId TransferProposal
        with newOwner: Party
        do
            assert (locker == owner)
            create TransferProposal
             with coin=this; newOwner

      --Lock can be done if owner decides to bring a locker on board
      Lock : ContractId LockableCoin
        with newLocker: Party
        do
          assert (newLocker /= owner)
          create this with locker = newLocker

    --Unlock only makes sense if the coin is in locked state
    controller locker can
      Unlock
        : ContractId LockableCoin
        do
```

(continues on next page)

(continued from previous page)

```
            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

    controller owner can

      Transfer : ContractId TransferProposal
        with newOwner: Party
        do
            create TransferProposal
             with coin=this; newOwner
```

(continues on next page)

(continued from previous page)

```
    --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.
    controller issuer can
      Archives
        : ()
        do assert (issuer == owner)
```

There is no need to make a change to the original contract. With two additional contracts, we can transfer the *Coin* ownership to a locker party.

Introduce a separate contract template *LockRequest* with the following features:

- LockRequest has a locker party as the single signatory, allowing the locker party to unilaterally initiate the process and specify locking terms.
- Once owner exercises *Accept* on the lock request, the ownership of coin is transferred to the locker.
- The *Accept* choice also creates a *LockedCoinV2* that represents *Coin* in locked state.

```
template LockRequest
  with
    locker: Party
    maturity: Time
    coin: Coin
  where
    signatory locker

    controller coin.owner can
      Accept : LockResult
        with coinCid : ContractId Coin
        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
```

(continues on next page)

(continued from previous page)

```
    controller locker can
      UnlockV2
        : ContractId Coin
        with coinCid : ContractId Coin
        do
          inputCoin <- fetch coinCid
          assert (inputCoin.owner == locker)
          tpCid <- exercise coinCid Transfer with newOwner = coin.owner
          exercise tpCid AcceptTransfer

    controller coin.owner can
      ClawbackV2
        : ContractId Coin
        with coinCid : ContractId Coin
        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
```

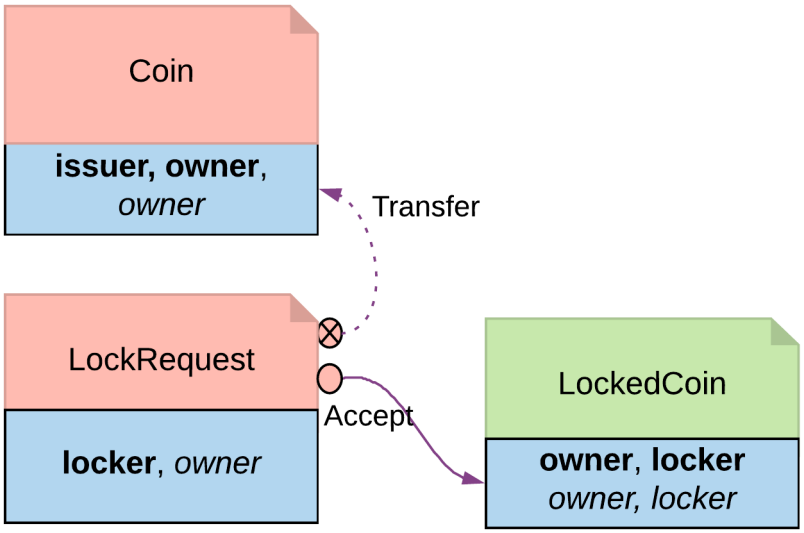


Fig. 7: Locking By Safekeeping Diagram

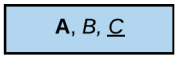
Trade-offs

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.

2.5.6 Diagram legends

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

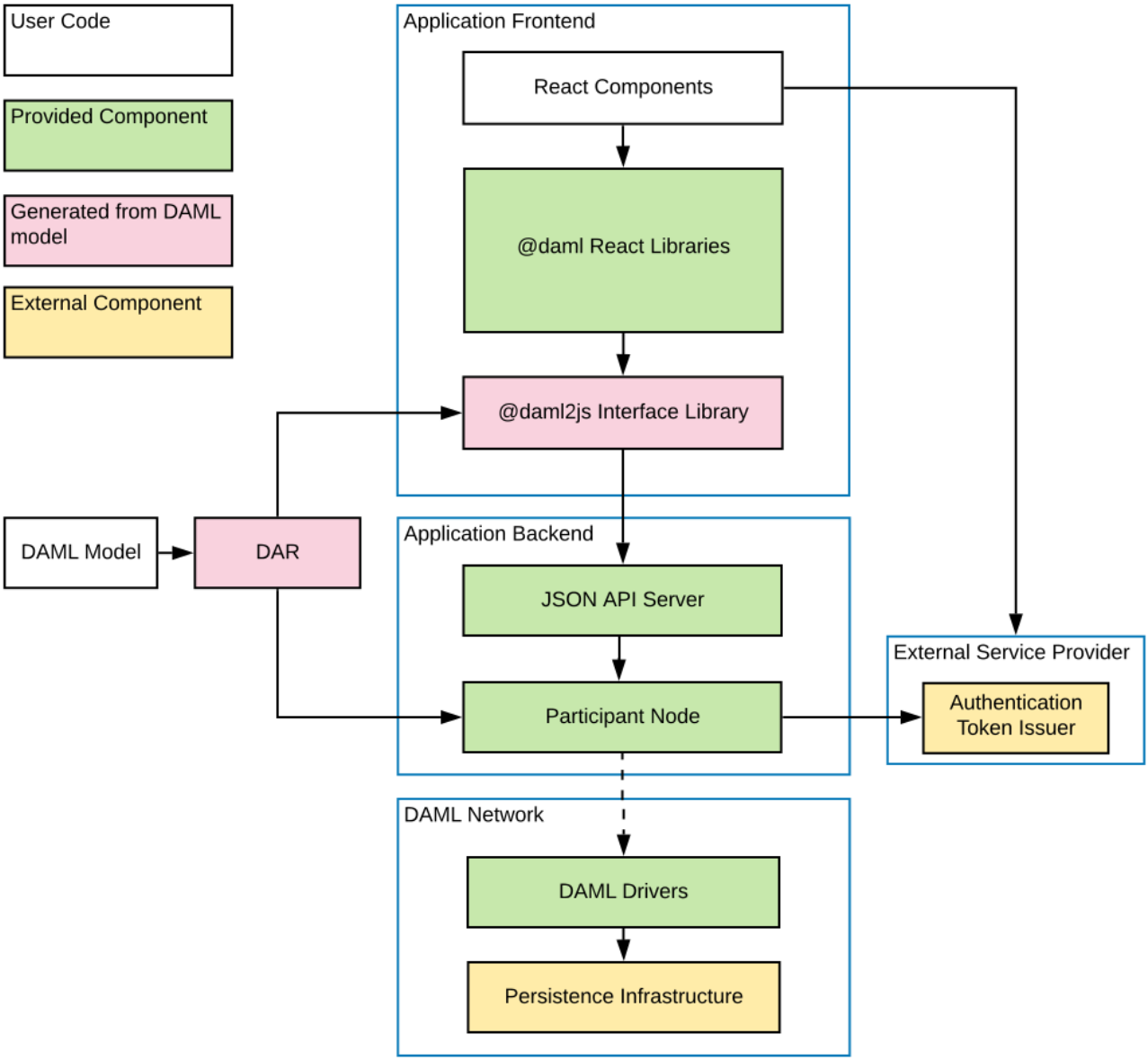
Reference to contractId

Chapter 3

Building applications

3.1 Application architecture

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



The above image shows the recommended architecture. Of course there are many ways how you can change the architecture and technology stack to fit your needs, which we'll mention in the corresponding sections. Note that the Participant Node is integrated into the Daml drivers in some cases rather than being part of the Application Backend. See *Daml Ecosystem Overview* for more details.

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

3.1.1 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

```
daml start --start-navigator=no
```

in the root of the project. 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. See *Deploying to Daml Ledgers* for an in depth manual for specific ledgers.

3.1.2 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 *Scala* 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.

3.1.3 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.

3.1.4 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.

3.1.4.1 Command deduplication

The interaction of a Daml application with the ledger is inherently asynchronous: applications send commands to the ledger, and some time later they see the effect of that command on the ledger.

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 the various 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 fall back to using the configured maximum deduplication period. The ledger will then guarantee that commands with the same *change ID* will be ignored within the deduplication period.

To use command deduplication, you should:

> Use generous values for the deduplication duration. It should be large enough such that you can assume the command was permanently lost if the deduplication period has passed and you still don't observe any effect of the command on the ledger (i.e. you don't see a transaction with the command ID via the *transaction service*).
> Make sure you set command IDs deterministically, that is to say: the same command must use the same command ID. This is useful for the recovery procedure after an application crash/restart, in which the application inspects the state of the ledger (e.g. via the *Active contracts service*) and sends commands to the ledger. When using deterministic command IDs, any commands that had been sent before the application restart will be discarded by the ledger to avoid duplicate submissions.
> If you are not sure whether a command was submitted successfully, just resubmit it with the same deduplication duration. If the new command was submitted within the deduplication period, the duplicate submission will safely be ignored. If the deduplication period has passed, you can assume the command was lost or rejected and a new submission is justified.

For more details on command deduplication, see the *Ledger API Services* documentation.

3.1.4.2 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.

3.1.4.3 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*.

3.2 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.

3.2.1 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.

3.2.1.1 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.

3.2.1.2 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_1`,` τ_2`)`	`Tuple`$_2$`<`τ_1`,` τ_2`>`	`{_1:` τ_1`; _2:` τ_2`}`

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**).

3.2.1.3 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 }
```

(continues on next page)

(continued from previous page)

```
  |  { 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.

```
import Ledger from  '@daml/ledger';
import {Iou, Transfer} from /* ... */;

const ledger = new Ledger(/* ... */);

// Contract creation; Bank issues Alice a USD $1MM IOU.

const iouDetails: Iou = {
  issuer: 'Chase',
  owner: 'Alice',
  currency: 'USD',
  amount: 1000000.0,
};
const aliceIouCreateEvent = await ledger.create(Iou, iouDetails);
const aliceIouContractId = aliceIouCreateEvent.contractId;

// Choice execution; Alice transfers ownership of the IOU to Bob.

const transferDetails: Transfer = {
  newOwner: 'Bob',
```

(continues on next page)

The `undefined` type parameter captures the fact that `Iou` has no contract key.

(continued from previous page)

```
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.

3.2.2 @daml/react

@daml/react documentation

3.2.3 @daml/ledger

@daml/ledger documentation

3.2.4 @daml/types

@daml/types documentation

3.2.5 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.

3.2.5.1 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.

3.2.5.2 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`.

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.

3.2.5.3 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.

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.

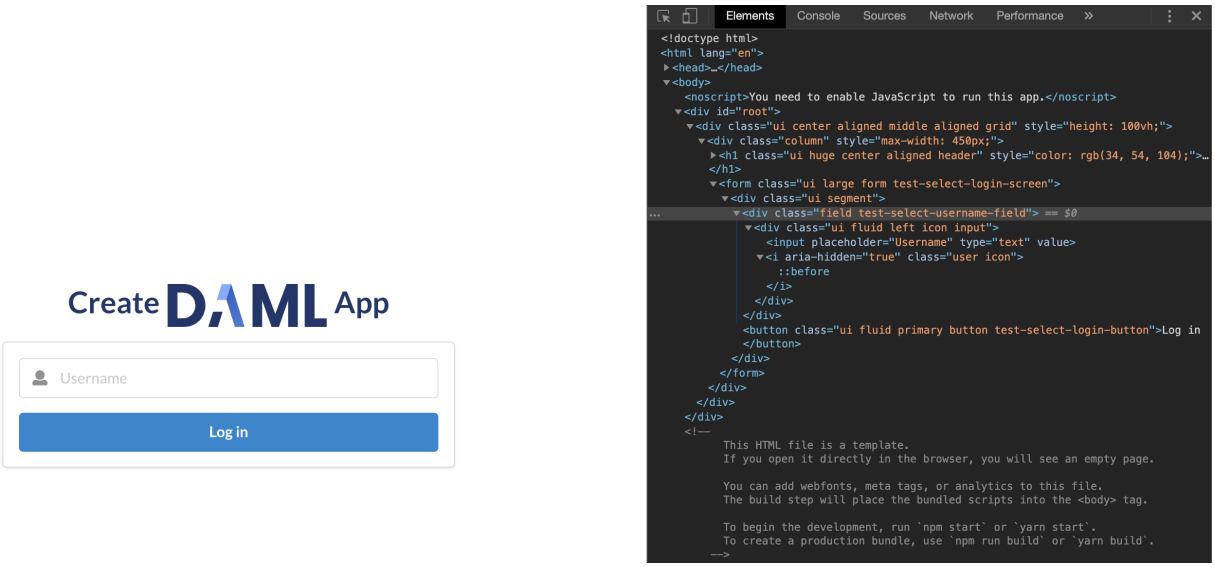
The strings used to find UI elements, e.g. `'.test-select-username-field'` and `'.test-select-login-button'`, are CSS Selectors. You may have seen them before in CSS styling of web pages. In this case we use *class selectors*, which look for CSS classes we've given to elements in our React components.

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.

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.

3.2.5.4 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.

3.2.5.5 The Full Test Suite

3.3 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.

3.3.1 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.

3.3.1.1 ContractId

Contract ids are expressed as their string representation:

```
"123"
"XYZ"
"foo:bar#baz"
```

3.3.1.2 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^{38}-1) 10^{10}, (10^{38}-1) 10^{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"
9999999999999999999999999999990
+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.

3.3.1.3 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`.

3.3.1.4 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.

3.3.1.5 Party

Represented using their string representation, without any additional quotes:

```
"Alice"
"Bob"
```

3.3.1.6 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.

3.3.1.7 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].

3.3.1.8 Text

Represented as strings.

3.3.1.9 Bool

Represented as booleans.

3.3.1.10 Record

Input

Records can be represented in two ways. As objects:

```
{ f1: v1, ..., f□: v□ }
```

And as arrays:

```
[ v1, ..., 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.

3.3.1.11 List

Lists are represented as

```
[v1, ..., v□]
```

3.3.1.12 TextMap

TextMaps are represented as objects:

```
{ k1: v1, ..., k□: v□ }
```

3.3.1.13 GenMap

GenMaps are represented as lists of pairs:

```
[ [k1, v1], [k□, v□] ]
```

Order does not matter. However, any duplicate keys will cause the map to be treated as invalid.

3.3.1.14 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
{ 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 `[f1, ..., 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 }`.

3.3.1.15 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.

3.3.1.16 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`.

3.3.2 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.

3.3.2.1 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

3.3.2.2 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.

3.3.2.3 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.

3.3.2.4 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.

3.3.2.5 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 the Enterprise Edition 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

3.3.3 Metrics

3.3.3.1 Enable and configure reporting

To enable metrics and configure reporting, you can use the two following CLI options:

`--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 sandbox 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.

3.3.3.2 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 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).

3.3.3.3 List of metrics

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

`daml.http_json_api.command_submission_timing`

A timer. Meters how long processing of a command submission request takes

`daml.http_json_api.query_all_timing`

A timer. Meters how long processing of a query GET request takes

`daml.http_json_api.query_matching_timing`

A timer. Meters how long processing of a query POST request takes

`daml.http_json_api.fetch_timing`

A timer. Meters how long processing of a fetch request takes

`daml.http_json_api.get_party_timing`

A timer. Meters how long processing of a get party/parties request takes

`daml.http_json_api.allocate_party_timing`

A timer. Meters how long processing of a party management request takes

`daml.http_json_api.download_package_timing`

A timer. Meters how long processing of a package download request takes

`daml.http_json_api.upload_package_timing`

A timer. Meters how long processing of a package upload request takes

`daml.http_json_api.incoming_json_parsing_and_validation_timing`

A timer. Meters how long parsing and decoding of an incoming json payload takes

`daml.http_json_api.response_creation_timing`

A timer. Meters how long the construction of the response json payload takes

`daml.http_json_api.response_creation_timing`

A timer. Meters how long the construction of the response json payload takes

`daml.http_json_api.db_find_by_contract_key_timing`

A timer. Meters how long a find by contract key database operation takes

`daml.http_json_api.db_find_by_contract_id_timing`

A timer. Meters how long a find by contract id database operation takes

`daml.http_json_api.command_submission_ledger_timing`

A timer. Meters how long processing of the command submission request takes 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

3.3.4 Running the JSON API

3.3.4.1 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 ./.daml/dist/quickstart-
↪0.0.1.dar
```

3.3.4.2 Start the HTTP JSON API Service

Basic

The most basic way to start the JSON API is with the command:

```
daml json-api --ledger-host localhost --ledger-port 6865 --http-port 7575
```

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.

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-1.5.0.jar --ledger-host localhost --ledger-port 6865 --
↪http-port 7575
```

Replace the version number `1.5.0` 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 use the `--query-store-jdbc-config` flag, an example of which is below.

Note: When you use the Query Store you'll want your first run to specify `createSchema=true` so that all the necessary tables are created. After the first run make sure `createSchema=false` so that it doesn't attempt to create the tables again.

```
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,createSchema=false"
```

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

3.3.4.3 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 the format *described in the Token Payload section here*. 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` (note that before SDK 1.7.0, every request required exactly one party and before SDK 1.8.0 actAs was limited to exactly one party). In addition to that, 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"]
  }
}
```

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.
↪eyJodHRwczovL2RhbWwuY29tL2xlZGdlci1hcGkiOnsibGVkZ2VySWQiOiJNeUxlZGdlciIsImFwcGxpY
↪34zzF_fbWv7p60r5s1kKzwndvGdsJDX-W4Xhm4oVdpk
```

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

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJodHRwczovL2RhbWwuY29tL2xlZGdlci1hcGkiOnsibGVkZ2VySWQiOiJNeUxlZGdlciIsImFwcGxpY
↪0uPPZtM1AmKvnGixt_Qo53cMDcpnziCjKKiWLvMX2VM
```

Auth via HTTP

Set HTTP header `Authorization: Bearer paste-jwt-here`

Example:

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJodHRwczovL2RhbWwuY29tL2xlZGdlci1hcGkiOnsibGVkZ2VySWQiOiJNeUxlZGdlciIsImFwcGxpY
↪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.
↪eyJodHRwczovL2RhbWwuY29tL2xlZGdlci1hcGkiOnsibGVkZ2VySWQiOiJNeUxlZGdlciIsImFwcGxpY
↪34zzF_fbWv7p60r5s1kKzwndvGdsJDX-W4Xhm4oVdpk
```

3.3.5 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, 404)
3. failure due to a server-side problem (500)

The **JSON API** can return one of the following HTTP status codes:

200 - OK
400 - Bad Request (Client Error)
401 - Unauthorized, authentication required
404 - Not Found
500 - Internal Server Error

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 | 404 | 500>,
    "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.

3.3.5.1 Successful response, HTTP status: 200 OK

Content-Type: `application/json`
Content:

```
{
    "status": 200,
    "result": <JSON object>
}
```

3.3.5.2 Successful response with a warning, HTTP status: 200 OK

Content-Type: `application/json`
Content:

```
{
    "status": 200,
```

(continues on next page)

(continued from previous page)

```
    "result": <JSON object>,
    "warnings": <JSON object>
}
```

3.3.5.3 Failure, HTTP status: 400 | 401 | 404 | 500

Content-Type: `application/json`
Content:

```
{
    "status": <400 | 401 | 404 | 500>,
    "errors": <JSON array of strings>
}
```

3.3.5.4 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"]}
```

3.3.6 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]
```

3.3.6.1 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*.

3.3.6.2 HTTP Response

Content-Type: `application/json`
Content:

```
{
    "status": 200,
    "result": {
        "observers": [],
        "agreementText": "",
        "payload": {
            "observers": [],
            "issuer": "Alice",
            "amount": "999.99",
            "currency": "USD",
            "owner": "Alice"
```

(continues on next page)

(continued from previous page)

```
        },
        "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).

3.3.7 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* used when submitting a command to the ledger.

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

Where:

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

3.3.8 Exercise by Contract ID

The JSON command below, demonstrates how to exercise an `Iou_Transfer` choice on an `Iou` contract:

```
    controller owner can
      Iou_Transfer : ContractId IouTransfer
        with
          newOwner : Party
        do create IouTransfer with iou = this; newOwner
```

3.3.8.1 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).

3.3.8.2 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": "",
```

(continues on next page)

(continued from previous page)

```
                        "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.

3.3.9 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
```

3.3.9.1 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.

3.3.9.2 HTTP Response

Formatted similar to *Exercise by Contract ID response*.

3.3.10 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.

3.3.10.1 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).

3.3.10.2 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": {
```

(continues on next page)

(continued from previous page)

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

3.3.11 Fetch Contract by Contract ID

3.3.11.1 HTTP Request

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

application/json body:

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

3.3.11.2 Contract Not Found HTTP Response

Content-Type: `application/json`
Content:

```
{
    "status": 200,
```

(continues on next page)

(continued from previous page)

```
    "result": null
}
```

3.3.11.3 Contract Found HTTP Response

Content-Type: `application/json`
Content:

```
{
    "status": 200,
    "result": {
        "observers": [],
        "agreementText": "",
        "payload": {
            "iou": {
                "observers": [],
                "issuer": "Alice",
                "amount": "999.99",
                "currency": "USD",
                "owner": "Alice"
            },
            "newOwner": "Alice"
        },
        "signatories": [
            "Alice"
        ],
        "contractId": "#201:1",
        "templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:IouTransfer
↪"
    }
}
```

3.3.12 Fetch Contract by Key

3.3.12.1 HTTP Request

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

```
{
    "templateId": "Account:Account",
    "key": {
        "_1": "Alice",
        "_2": "abc123"
    }
}
```

3.3.12.2 Contract Not Found HTTP Response

Content-Type: `application/json`
Content:

```
{
    "status": 200,
    "result": null
}
```

3.3.12.3 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
↪"
    }
}
```

3.3.13 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.

3.3.13.1 HTTP Request

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

3.3.13.2 HTTP Response

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

3.3.14 Get all Active Contracts Matching a Given Query

List currently active contracts that match a given query.

3.3.14.1 HTTP Request

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

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

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*.

3.3.14.2 Empty HTTP Response

Content-Type: `application/json`
Content:

```
{
    "status": 200,
    "result": []
}
```

3.3.14.3 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.

3.3.14.4 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
}
```

3.3.15 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"
  ]
}
```

3.3.15.1 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",
```

(continues on next page)

(continued from previous page)

```
      "isLocal": true
    }
  ]
}
```

Please note that the order of the party objects in the response is not guaranteed to match the order of the passed party identifiers.

Where

`identifier` – a stable unique identifier of a Daml party,
`displayName` – optional human readable name associated with the party. Might not be unique,
`isLocal` – true if party is hosted by the backing participant.

3.3.15.2 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.

3.3.16 Fetch All Known Parties

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

3.3.16.1 HTTP Response

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

3.3.17 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.

3.3.17.1 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:

```
{}
```

3.3.17.2 HTTP Response

```
{
  "result": {
    "identifier": "Carol",
    "displayName": "Carol & Co. LLC",
    "isLocal": true
  },
  "status": 200
}
```

3.3.18 List All DALF Packages

3.3.18.1 HTTP Request

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

3.3.18.2 HTTP Response

```
{
  "result": [
    "c1f1f00558799eec139fb4f4c76f95fb52fa1837a5dd29600baa1c8ed1bdccfd",
    "733e38d36a2759688a4b2c4cec69d48e7b55ecc8dedc8067b815926c917a182a",
    "bfcd37bd6b84768e86e432f5f6c33e25d9e7724a9d42e33875ff74f6348e733f",
    "40f452260bef3f29dede136108fc08a88d5a5250310281067087da6f0baddff7",
    "8a7806365bbd98d88b4c13832ebfa305f6abaeaf32cfa2b7dd25c4fa489b79fb"
  ],
  "status": 200
}
```

Where `result` is the JSON array containing the package IDs of all loaded DALFs.

3.3.19 Download a DALF Package

3.3.19.1 HTTP Request

URL: `/v1/packages/<package ID>`
Method: `GET`
Content: <EMPTY>

Note that the desired package ID is specified in the URL.

3.3.19.2 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.

3.3.19.3 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
}
```

3.3.20 Upload a DAR File

3.3.20.1 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.

3.3.20.2 HTTP Response, status: 200 OK

Content-Type: `application/json`
Content:

```
{
    "result": 1,
    "status": 200
}
```

3.3.20.3 HTTP Response with Error

Content-Type: `application/json`
Content:

```
{
    "errors": [
        "io.grpc.StatusRuntimeException: INVALID_ARGUMENT: Invalid
↪argument: Invalid DAR: package-upload, content: []}]"
    ],
    "status": 500
}
```

3.3.21 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.
↪eyJodHRwczovL2RhbWwuY29tL2xlZGdlci1hcGkiOnsibGVkZ2VySWQiOiJNeUxlZGdlciIsImFwcGxpY
↪34zzF_fbWv7p60r5s1kKzwndvGdsJDX-W4Xhm4oVdp";
const subprotocols = [`${tokenPrefix}${jwt}`, wsProtocol];

const ws = new WebSocket("ws://localhost:7575/v1/stream/query",
↪subprotocols);

ws.addEventListener("open", function open() {
  ws.send(JSON.stringify({templateIds: ["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.

3.3.21.1 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
}
```

3.3.21.2 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",
                "owner": "Alice"
            },
            "signatories": ["Alice"],
            "contractId": "#1:0",
            "templateId":
↪"eb3b150383a979d6765b8570a17dd24ae8d8b63418ee5fd20df20ad2a1c13976:Iou:Iou
↪"
        },
        "matchedQueries": [1, 2]
    }]
}
```

where `matchedQueries` indicates the 0-based indices into the request list of queries that matched this contract.

Every `events` block following the end of contracts that existed when the request started includes an `offset`. The stream is guaranteed to send an offset immediately at the beginning of this live data, which may or may not contain any `events`; if it does not contain events and no events were emitted before, it may be `null` if there was no transaction on the ledger or a string representing the current ledger end; otherwise, it will be a string. For example, you might use it to turn off an initial loading indicator:

```
{
    "events": [],
    "offset": "2"
}
```

Note: Events in the following live data may include `events` that precede this `offset` if an earlier

per-query `offset` was specified.

This has been done with the intent of allowing to use per-query `offset` s to efficiently use a single connection to multiplex various requests. To give an example of how this would work, let's say that there are two contract templates, `A` and `B` . Your application first queries for `A` s without specifying an offset. Then some client-side interaction requires the application to do the same for `B` s. The application can save the latest observed offset for the previous query, which let's say is `42`, and issue a new request that queries for all `B` s without specifying an offset and all `A` s from `42`. While this happens on the client, a few more `A` s and `B` s are created and the new request is issued once the latest offset is `47`. The response to this will contain a message with all active `B` s, followed by the message reporting the offset `47`, followed by a stream of live updates that contains new `A` s starting from `42` and new `B` s starting from `47` .

To keep the stream alive, you'll occasionally see messages like this, which can be safely ignored if you do not need to capture the last seen ledger offset:

```
{"events":[],"offset":"5609"}
```

where `offset` is the last seen ledger offset.

After submitting an `Iou_Split` exercise, which creates two contracts and archives the one above, the same stream will eventually produce:

```
{
    "events": [{
        "archived": {
            "contractId": "#1:0",
            "templateId":
↪"eb3b150383a979d6765b8570a17dd24ae8d8b63418ee5fd20df20ad2a1c13976:Iou:Iou
↪"
        }
    }, {
        "created": {
            "observers": [],
            "agreementText": "",
            "payload": {
                "observers": [],
                "issuer": "Alice",
                "amount": "42.42",
                "currency": "USD",
                "owner": "Alice"
            },
            "signatories": ["Alice"],
            "contractId": "#2:1",
            "templateId":
↪"eb3b150383a979d6765b8570a17dd24ae8d8b63418ee5fd20df20ad2a1c13976:Iou:Iou
↪"
        },
        "matchedQueries": [0, 2]
    }, {
        "created": {
```

(continues on next page)

(continued from previous page)

```
            "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.

3.3.21.3 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.

3.3.22 Healthcheck Endpoints

The HTTP JSON API provides two healthcheck endpoints for integration with schedulers like Kubernetes.

3.3.22.1 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.

3.3.22.2 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.

3.4 Daml Script

3.4.1 Daml Script Library

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

3.4.1.1 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 *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, ()))

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 : 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.

Daml scenarios provide a simple way for testing Daml models and getting quick feedback in Daml studio. However, scenarios are run in a special process and do not interact with an actual ledger. This means that you cannot use scenarios to test other ledger clients, e.g., your UI or *Daml triggers*.

Daml Script addresses this problem by providing you with an API with the simplicity of Daml scenarios and all the benefits such as being able to reuse your Daml types and logic while running against an actual ledger in addition to allowing you to experiment in *Daml Studio*. 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`

3.4.2 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
```

(continues on next page)

(continued from previous page)

```
  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. `Commands` is similar to `Update` which is used in the `submit` function in scenarios. 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 ()
```

3.4.2.1 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.

```
  alice <- allocateParty "Alice"
  bob <- allocateParty "Bob"
  bank <- allocateParty "Bank"
  let parties = LedgerParties bank alice bob
```

We now call the `initialize` function that we defined before on the parties that we have just allocated.

```
  initialize parties
```

Another option for getting access to the relevant party ids is to use `listKnownParties` to pick out the party with a given display name. This is mainly useful in demo scenarios because display names are not guaranteed to be unique.

```
queryParties : Script (Optional LedgerParties)
queryParties = do
  knownParties <- listKnownParties
  pure $ do
    bank <- party <$> find (\p -> p.displayName == Some "Bank")
↪knownParties
    alice <- party <$> find (\p -> p.displayName == Some "Alice")
↪knownParties
    bob <- party <$> find (\p -> p.displayName == Some "Bob") knownParties
    pure LedgerParties{..}

initializeFromQuery : Script ()
initializeFromQuery = do
  optParties <- queryParties
  case optParties of
    None -> fail "Could not find parties with correct display names"
    Some parties -> initialize parties
```

3.4.2.2 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)
```

(continues on next page)

(continued from previous page)

```
  aliceCoins <- query @Coin alice
  assertEq [Coin bank alice] (map snd aliceCoins)

  bobCoins <- query @Coin bob
  assertEq [Coin bank bob] (map snd bobCoins)
```

3.4.2.3 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 scenarios 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 usfeful if the given program has a type `Script b`, where `b` is a meaningful value, which is not the case here: all of our `Script` programs have type `Script ()`.

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:

```
{
  "alice": "Alice",
  "bob": "Bob",
  "bank": "Bank"
}
```

Using that file, we can initialize our ledger passing it in via `--input-file`:

```
daml script --dar .daml/dist/script-example-0.0.1.dar --script-name
ScriptExample:initialize --ledger-host localhost --ledger-port 6865 --
input-file ledger-parties.json
```

If you open Navigator, you can now see the contracts that have been created.

3.4.3 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:initializeFixed` field in your `daml.yaml`. This will automatically be picked up by `daml start` and used to initialize sandbox. Since it is often useful to create a party with a specific party identifier during development, you can use the `allocatePartyWithHint` function which accepts not only the display name but also a hint for the party identifier. On Sandbox, the hint will be used directly as the party identifier of the newly allocated party. This allows us to implement `initializeFixed` as a small wrapper around the `initialize` function we defined above:

```
initializeFixed : Script ()
initializeFixed = do
  bank <- allocatePartyWithHint "Bank" (PartyIdHint "Bank")
  alice <- allocatePartyWithHint "Alice" (PartyIdHint "Alice")
  bob <- allocatePartyWithHint "Bob" (PartyIdHint "Bob")
  let parties = LedgerParties{..}
  initialize parties
```

3.4.3.1 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 and Daml for PostgreSQL, other 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`.

3.4.4 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"}
    },
    "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.

3.4.5 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.

3.4.6 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.

3.5 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.

3.5.1 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 .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`.

3.5.2 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)
```

3.5.3 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
```

3.5.4 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.

3.5.5 Connection to a Ledger with Authorization

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

3.5.6 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"}]
```

3.6 Upgrading and Extending Daml applications

3.6.1 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.

3.6.2 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.

3.6.2.1 Daml upgrade overview

In a Daml application running on a distributed ledger, the signatories of a contract have agreed to one specific version of a template. Changing the definition of a template, e.g., by extending it with a new data field or choice without agreement from its signatories would completely break the authorization guarantees provided by Daml.

Therefore, Daml takes a different approach to upgrades and extensions. Rather than having a separate concept of data migration that sidesteps the fundamental guarantees provided by Daml, *upgrades are expressed as Daml contracts*. This means that the same guarantees and rules that apply to other Daml contracts also apply to upgrades.

In a Daml application, it thus makes sense to think of upgrades as an *extension of an existing application* instead of an operation that replaces existing contracts with a newer version. The existing templates stay on the ledger and can still be used. Contracts of existing templates are not automatically replaced by newer versions. However, the application is extended with new templates. Then if all signatories of a contract agree, a choice can archive the old version of a contract and create a new contract instead.

3.6.2.2 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"
```

3.6.2.3 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
```

Now we can build and deploy `carbon-1.0.0`.

```
$ cd example/carbon-1.0.0
$ daml build
$ daml ledger upload-dar --port 6865
```

3.6.2.4 Create some carbon-1.0.0 certificates

Let's create some certificates!

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.
    3. Create a 2nd proposal in the same way.
2. **Login as Bob:**
    1. Exercise the *CarbonCertProposal_Accept* choice on both proposal contracts.

3.6.2.5 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 cryptographically computed package identifier would change. Consequently, it would not match the package identifier of the original `CarbonCert` contracts from `carbon-1.0.0` which are live on the ledger.

Here is the new project config:

```
name: carbon
version: 2.0.0
dependencies:
  - daml-prim
  - daml-stdlib
```

Now we can build and deploy `carbon-2.0.0`.

```
$ cd example/carbon-2.0.0
$ daml build
$ daml ledger upload-dar --port 6865
```

3.6.2.6 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
```

3.6.2.7 Upgrade existing certificates from carbon-1.0.0 to carbon-2.0.0

We start the navigator again.

```
$ cd example/carbon-upgrade
$ daml navigator server localhost 6865
```

Finally, we point a browser to http://localhost:4000 and can start the carbon certificates upgrades:

1. **Login as Alice**
    1. Select Templates tab.
    2. Create an `UpgradeCarbonCertProposal` with Alice as issuer and Bob as owner.
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.

3.6.2.8 Further Steps

For the upgrade of our carbon certificate model above, we performed all steps manually via Navigator. However, if Alice had issued millions of carbon certificates, performing all upgrading steps manually becomes infeasible. It thus becomes necessary to automate these steps. We will go through a potential implementation of an automated upgrade in the *next section*.

3.6.3 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.

3.6.3.1 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`.

3.6.3.2 Implementation of the Daml Script

In our Daml Script, we are first going to query the ACS (Active Contract Set) to find all `CarbonCert` contracts issued by us. Next, we are going to extract the owner of each of those contracts and remove any duplicates coming from multiple certificates issued to the same owner. Finally, we iterate over the owners and create an `UpgradeCarbonCertAgreement` contract for each owner.

```
initiateUpgrade : Party -> Script ()
initiateUpgrade issuer = do
  certs <- query @CarbonCert issuer
  let myCerts = filter (\(_cid, c) -> c.issuer == issuer) certs
  let owners = dedup $ map (\(_cid, c) -> c.owner) myCerts
```

(continues on next page)

(continued from previous page)

```
  forA_ owners $ \owner -> do
    debug ("Creating upgrade proposal for: " <> show owner)
    submit issuer $ createCmd (UpgradeCarbonCertProposal issuer owner)
```

3.6.3.3 Implementation of the Daml Trigger

Our trigger does not need any custom user state and no heartbeat so the only interesting field in its definition is the rule.

```
upgradeTrigger : Trigger ()
upgradeTrigger = Trigger with
  initialize = pure ()
  updateState = \_msg -> pure ()
  registeredTemplates = AllInDar
  heartbeat = None
  rule = triggerRule
```

In our rule, we first filter out all agreements and certificates issued by us. Next, we iterate over all agreements. For each agreement we filter the certificates by the owner of the agreement and finally upgrade the certificate by exercising the `Upgrade` choice. We mark the certificate as pending which temporarily removes it from the ACS and therefore stops the trigger from trying to upgrade the same certificate multiple times if the rule is triggered in quick succession.

```
triggerRule : Party -> TriggerA () ()
triggerRule issuer = do
  agreements <-
    filter (\(_cid, agreement) -> agreement.issuer == issuer) <$>
    query @UpgradeCarbonCertAgreement
  allCerts <-
    filter (\(_cid, cert) -> cert.issuer == issuer) <$>
    query @CarbonCert
  forA_ agreements $ \(agreementCid, agreement) -> do
    let certsForOwner = filter (\(_cid, cert) -> cert.owner == agreement.
↪owner) allCerts
    forA_ certsForOwner $ \(certCid, _) ->
      emitCommands
        [exerciseCmd agreementCid (Upgrade certCid)]
        [toAnyContractId certCid]
```

The trigger is a long-running process and the rule will be executed whenever the state of the ledger changes. So whenever an owner accepts an upgrade proposal, the trigger will run the rule and upgrade all certificates of that owner.

3.6.3.4 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 .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.

```
setup : Script ()
setup = do
  alice <- allocatePartyWithHint "Alice" (PartyIdHint "Alice")
  bob <- allocatePartyWithHint "Bob" (PartyIdHint "Bob")
  charlie <- allocatePartyWithHint "Charlie" (PartyIdHint "Charlie")
  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 ()
```

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 --
↪wall-clock-time
```

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-party=Alice --wall-clock-time
```

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 `Party`. We can pass this in via the `--input-file` argument which we will point to a file `party.json` containing `"Alice"`. This allows us to change the party without having to change the code of the script.

```
$ 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 --wall-clock-time --input-file=party.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.

3.7 Authorization

When developing Daml applications using SDK tools, your local setup will most likely not verify any authorization - by default, any valid ledger API request will be accepted by the sandbox.

To run your application against a *deployed ledger*, you will need to add authorization.

3.7.1 Introduction

The *Ledger API* is used to request changes to the ledger (e.g., *Alice wants to exercise choice X on contract Y*), or to read data from the ledger (e.g., *Alice wants to see all active contracts*).

What requests are valid is defined by *integrity* and *privacy* parts the *Daml Ledger Model*. This model is defined in terms of *Daml parties*, and does not require any cryptographic information to be sent along with requests.

In particular, this model does not talk about authentication (*Is the request claiming to come from Alice really sent by Alice?*) and authorization (*Is Alice authorized to add a new Daml package to the ledger?*).

In practice, Daml ledgers will therefore need to add authentication to the ledger API.

Note: Depending on the ledger topology, a Daml ledger may consist of multiple participant nodes, each having its own ledger API server. Each participant node typically hosts different Daml parties, and only sees data visible to the parties hosted on that node (as defined by the Daml privacy model).

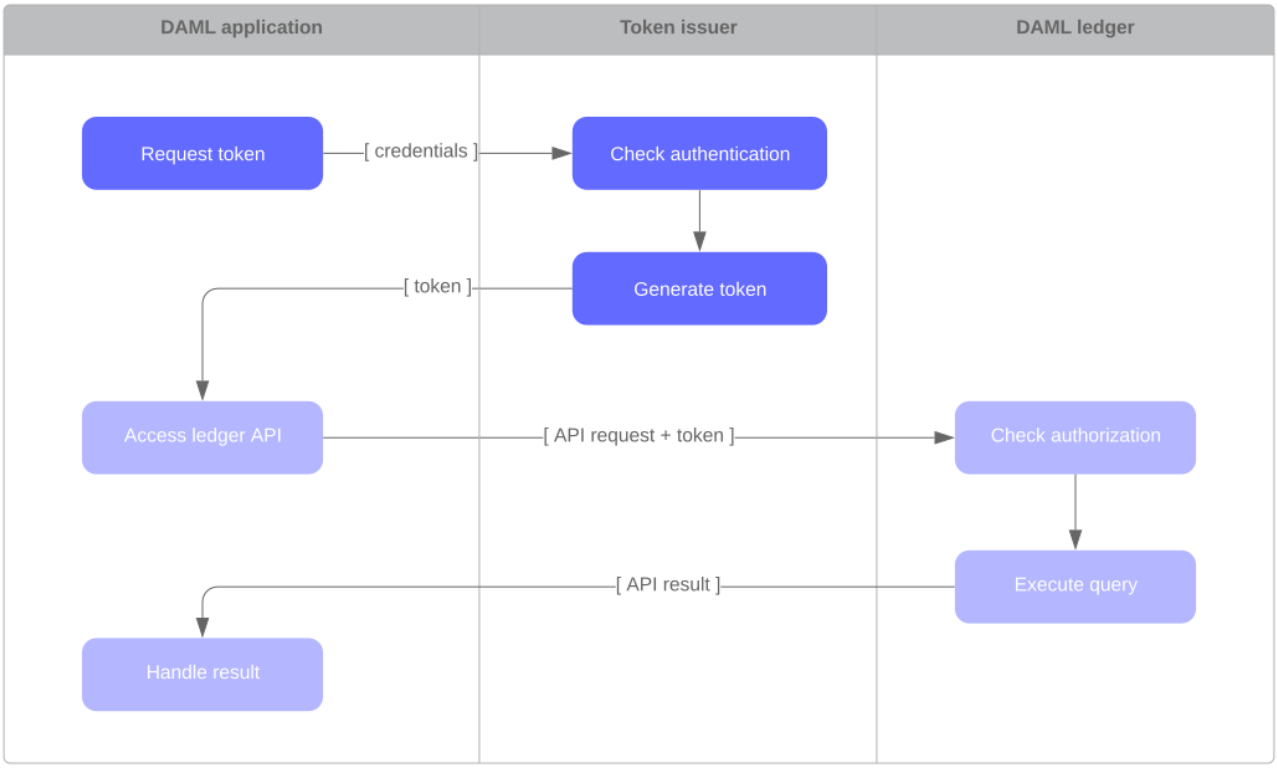
For more details on Daml ledger topologies, refer to the *Daml Ledger Topologies* documentation.

3.7.1.1 Adding authentication

How authentication is set up on a particular ledger is defined by the ledger operator. However, most authentication setups share the following pattern:

First, the Daml application contacts a token issuer to get an access token. The token issuer verifies the identity of the requesting user (e.g., by checking the username/password credentials sent with the request), looks up the privileges of the user, and generates a signed access token describing those privileges.

Then, the Daml application sends the access token along with each ledger API request. The Daml ledger verifies the signature of the token (to make sure it has not been tampered with), and then checks that the privileges described in the token authorize the given ledger API request.



Glossary:

> `Authentication` is the process of confirming an identity.
> `Authorization` is the process of checking permissions to access a resource.
> A `token` (or `access token`) is a tamper-proof piece of data that contains security information, such as the user identity or its privileges.
> A `token issuer` is a service that generates tokens. Also known as authentication server or Identity and Access Management (IAM) system .

3.7.2 Access tokens and claims

Access tokens contain information about the capabilities held by the bearer of the token. This information is represented by a *claim* to a given capability.

The claims can express the following capabilities:

> `public`: ability to retrieve publicly available information, such as the ledger identity

`admin`: ability to interact with admin-level services, such as package uploading and party allocation
`canReadAs(p)`: ability to read information off the ledger (like the active contracts) visible to the party `p`
`canActsAs(p)`: same as `canReadAs(p)`, with the added ability of issuing commands on behalf of the party `p`

The following table summarizes what kind of claim is required to access each Ledger API endpoint:

Ledger API service	Endpoint	Required claim
LedgerIdentityService	GetLedgerIdentity	public
ActiveContractsService	GetActiveContracts	for each requested party p: canReadAs(p)
CommandSubmissionService	Submit	for submitting party p: canActAs(p)
	CompletionEnd	public
	CompletionStream	for each requested party p: canReadAs(p)
CommandService	All	for submitting party p: canActAs(p)
LedgerConfigurationService	GetLedgerConfiguration	public
PackageService	All	public
PackageManagementService	All	admin
PartyManagementService	All	admin
PruningService	All	admin
ResetService	All	admin
TimeService	GetTime	public
	SetTime	admin
TransactionService	LedgerEnd	public
	All (except LedgerEnd)	for each requested party p: canReadAs(p)

Access tokens may be represented differently based on the ledger implementation.

To learn how these claims are represented in the Sandbox, read the *sandbox* documentation.

3.7.3 Getting access tokens

To learn how to receive access tokens for a deployed ledger, contact your ledger operator. This may be a manual exchange over a secure channel, or your application may have to request tokens at runtime using an API such as OAuth.

To learn how to generate access tokens for the Sandbox, read the *sandbox* documentation.

3.7.4 Using access tokens

To learn how to use access tokens in the Scala bindings, read the *Scala bindings authorization* documentation.

3.8 The Ledger API

3.8.1 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*.

3.8.1.1 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`.

3.8.1.2 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 of the submitting parties (i.e., the union of *party* and *act_as*), *application ID* and *command ID*.

Application-specific IDs

The following application-specific IDs, all of which are included in completion events, can be set in commands:

> A *submissionId*, returned to the submitting application only. It may be used to correlate specific submissions to specific completions.
> A *commandId*, returned to the submitting application only; it can be used to correlate commands to completions.
> A *workflowId*, 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 :ref‘change ID <change-id>‘:

> Applications can provide a *deduplication duration* for each command. If this parameter is not set, the default maximum deduplication period 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 :ref‘change ID <change-id>‘.
> Duplicate command submissions will be ignored until either the deduplication period of the original command has passed or the original submission was rejected (i.e. the command failed and resulted in a rejected transaction), 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.
> A command submission will return:
> - The result of the submission (`Empty` or a gRPC error), if the command was submitted outside of the deduplication period of a previous command with the same :ref‘change ID <change-id>‘ on the same participant.
> - The status error `ALREADY_EXISTS`, if the command was discarded by the ledger server because it was sent within the deduplication period of a previous command with the same :ref‘change ID <change-id>‘.
>
> If the ledger provides additional command deduplication across participants, the initial command submission might be successful, but ultimately the command can be rejected if the deduplication check fails on the ledger.

For details on how to use command deduplication, see the *Application Architecture Guide*.

Command completion service

Use the **command completion service** to find out the completion status of commands you have submitted.

Completions contain the `commandId` 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*.

3.8.1.3 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 *transactionId* (assigned by the server), the `workflowId`, the `commandId`, 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 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.

3.8.1.4 Utility services

Party management service

Use the **party management service** to allocate parties on the ledger and retrieve information about allocated parties.

Allocating parties is necessary to interact with the ledger. For more information, refer to the pages on *Identity Management* and *the API reference documentation*.

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

Use the **ledger identity service** to get the identity string of the ledger that your application is connected to.

You need to include this identity string when submitting commands. Commands with an incorrect identity string are 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.

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*.

3.8.1.5 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*.

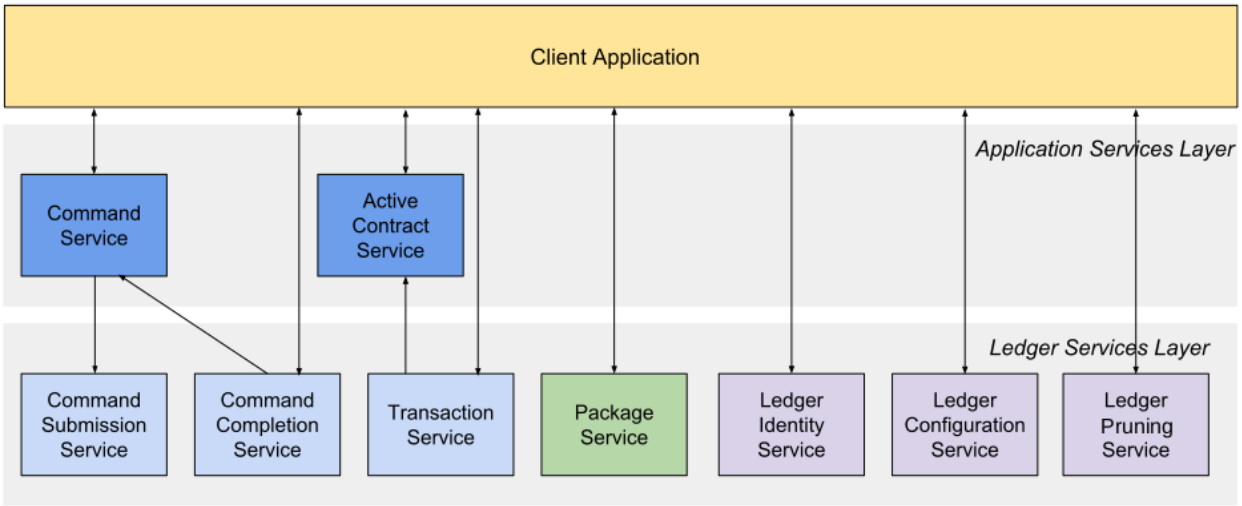
Reset service

Use the **reset service** to reset the ledger state, as a quicker alternative to restarting the whole ledger application.

This resets all state in the ledger, *including the ledger ID*, so clients will have to re-fetch the ledger ID from the identity service after hitting this endpoint.

For full details, see *the proto documentation for the service*.

3.8.1.6 Services diagram



3.8.2 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.

3.8.2.1 Getting started

You can get the protobufs from a GitHub release, or from the `daml` repository here.

3.8.2.2 Protobuf reference documentation

For full details of all of the Ledger API services and their RPC methods, see *Ledger API Reference*.

3.8.2.3 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.

3.8.2.4 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*.

3.8.2.5 Error handling

For the standard error codes that the server or the client might return, see the gRPC documentation .

For submitted commands, there are these response codes:

ABORTED The platform failed to record the result of the command due to a transient server-side error 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.

ALREADY_EXISTS The command was rejected because it was sent within the deduplication period of a previous command with the same change ID.

INVALID_ARGUMENT The submission failed because of a client error. The platform will definitely reject resubmissions of the same command.

OK, INTERNAL, UNKNOWN (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) Resubmit the command with the same command_id.

3.8.3 Ledger API Reference

3.8.3.1 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`). Required
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)

3.8.3.2 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

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. - `ABORTED`: if the request is rejected or 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. - `UNIMPLEMENTED`: if this method is not supported by the backing ledger.

3.8.3.3 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.

3.8.3.4 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.

Other GRPC errors can be returned depending on the type of condition preventing a prune: - `OUT_OF_RANGE`: if the participant is not yet able to prune at the specified offset, but without user intervention the offset will eventually be usable for pruning. - `FAILED_PRECONDITION` if some sort of user intervention is required before pruning can proceed at the specified offset.

3.8.3.5 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-sql: returns an identifier supplied on command line at launch time daml-on-kv-ledger: as above 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 - `INVALID_ARGUMENT`: if the provided hint and/or display name is invalid on the given ledger (see below). daml-on-sql: suggestion's uniqueness is checked and call rejected if the identifier is already present 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

3.8.3.6 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`). Required

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`). Required
application_id	*string*		Only completions of commands submitted with the same application_id will be visible in the stream. Must be a valid LedgerString (as described in `value.proto`). Required
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 or if the ledger has been pruned before `begin` - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields - `OUT_OF_RANGE`: if the absolute offset is not before 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

3.8.3.7 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

SubmitAndWaitForTransactionResponse

Field	Type	Label	Description
transaction	*Transaction*		The flat transaction that resulted from the submitted command. Required

SubmitAndWaitForTransactionTreeResponse

Field	Type	Label	Description
transaction	*TransactionTree*		The transaction tree that resulted from the submitted command. Required

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 - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields - `RESOURCE_EXHAUSTED`: if the number of in-flight commands reached the maximum (if a limit is configured) - `UNAVAILABLE`: if the participant is not yet ready to submit commands or if the service has been shut down. - `ABORTED`: if a contract key is missing or duplicated due to for example contention on resources
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 - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields - `RESOURCE_EXHAUSTED`: if the number of in-flight commands reached the maximum (if a limit is configured) - `UNAVAILABLE`: if the participant is not yet ready to submit commands or if the service has been shut down. - `ABORTED`: if a contract key is missing or duplicated due to for example contention on resources
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 - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields - `RESOURCE_EXHAUSTED`: if the number of in-flight commands reached the maximum (if a limit is configured) - `UNAVAILABLE`: if the participant is not yet ready to submit commands or if the service has been shut down. - `ABORTED`: if a contract key is missing or duplicated due to for example contention on resources
SubmitAndWaitForTransactionTree	*SubmitAndWaitRequest*	*SubmitAndWaitForTransactionTreeRe-*	Submits a single composite command, waits for its result, and returns the transaction tree. Propagates the gRPC error of failed submissions including Daml interpretation errors. Errors: - `UNAUTHENTICATED`:

3.8.3.8 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 - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields - `UNAVAILABLE`: if the participant is not yet ready to submit commands or if the service has been shut down. - `RESOURCE_EXHAUSTED`: if the participant or the ledger is overloaded. Clients should back off exponentially and retry. - `ABORTED`: if a contract key is missing or duplicated due to for example contention on resources

3.8.3.9 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
create	*CreateCommand*		
exercise	*ExerciseCommand*		
exerciseByKey	*ExerciseByKeyCommand*		
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`). Required
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 its part) that issued the command. This is used in tracing across different components and to let applications subscribe to their own submissions only. Must be a valid LedgerString (as described in `value.proto`). Required
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
deduplication_time	google.protobuf.Duration		Specifies the length of the deduplication period. Same semantics apply as for *deduplication_duration*.

Must be non-negative.

- 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.

  - min_ledger_time_abs
  - google.protobuf.Timestamp
  -
  - Lower bound for the ledger time assigned to the resulting transaction. Note: The ledger time of a transaction is assigned as part of command interpretation. Use this property if you expect that command interpretation will take a considerate amount of time, such that by the time the resulting transaction is sequenced, its assigned ledger time is not valid anymore. Must not be set at the same time as min_ledger_time_rel. Optional
  - min_ledger_time_rel
  - google.protobuf.Duration
  -
  - Same as min_ledger_time_abs, but specified as a duration, starting from the time the command is received by the server. Must not be set at the same time as min_ledger_time_abs. Optional
  - act_as
  - *string*
  - repeated
  - Set of parties 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 each of the given parties. This field supersedes the `party` field. The effective set of parties on whose behalf the command should be executed is the union of all parties listed in `party` and `act_as`, which must be non-empty. Each element must be a valid PartyIdString (as described in `value.proto`). Optional
  - read_as
  - *string*
  - repeated
  - Set of parties on whose behalf (in addition to all parties listed in `act_as`) contracts can be retrieved. This affects Daml operations such as `fetch`, `fetchByKey`, `lookupByKey`, `exercise`, and `exerciseByKey`. Note: A participant node of a Daml network can host multiple parties. Each contract present on the participant node is only visible to a subset of these parties. A command can only use contracts that are visible to at least one of the parties in `act_as` or `read_as`. This visibility check is independent from the Daml authorization rules for fetch operations. If ledger API authorization is enabled, then the authorization metadata must authorize the sender of the request to read contract data on behalf of each of the given parties. Optional
  - submission_id
  - *string*
  -
  - A unique identifier to distinguish completions for different submissions with the same change ID. Typically a random UUID. Applications are expected to use a different UUID for each retry of a submission with the same change ID. Must be a valid LedgerString (as described in `value.proto`).

If omitted, the participant or the committer may set a value of their choice. Optional

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

3.8.3.10 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 that was used for the submission, as described in `commands.proto`. Must be a valid LedgerString (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 for historic completions where this data is not available.
deduplication_offset	*string*		Specifies the start of the deduplication period by a completion stream offset.

Must be a valid LedgerString (as described in `value.proto`).

- deduplication_time
- google.protobuf.Duration
-
- Specifies the length of the deduplication period. It is measured in record time of completions.

Must be non-negative.

3.8.3.11 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
created	*CreatedEvent*		
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

3.8.3.12 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`). Required

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_time	google.protobuf.Duration		If a command submission specifies a deduplication period of length up to `max_deduplication_time`, the submission SHOULD not be rejected with `NOT_FOUND` because the deduplication period start is 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

3.8.3.13 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`). Required

LedgerIdentityService

Allows clients to verify that the server they are communicating with exposes the ledger they wish to operate on. Note that every ledger has a unique ID.

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

3.8.3.14 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
absolute	*string*		The format of this string is specific to the ledger and opaque to the client.
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.

3.8.3.15 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`). Required
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`). Required
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`). Required

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

3.8.3.16 com/daml/ledger/api/v1/testing/reset_service.proto

ResetRequest

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`). Required

ResetService

Service to reset the ledger state. The goal here is to be able to reset the state in a way that's much faster compared to restarting the whole ledger application (be it a sandbox or the real ledger server).

Note that *all* state present in the ledger implementation will be reset, most importantly including the ledger ID. This means that clients will have to re-fetch the ledger ID from the identity service after hitting this endpoint.

The semantics are as follows:

> When the reset service returns the reset is initiated, but not completed;
> While the reset is performed, the ledger will not accept new requests. In fact we guarantee that ledger stops accepting new requests by the time the response to Reset is delivered;
> In-flight requests might be aborted, we make no guarantees on when or how quickly this happens;
> The ledger might be unavailable for a period of time before the reset is complete.

Given the above, the recommended mode of operation for clients of the reset endpoint is to call it, then call the ledger identity endpoint in a retry loop that will tolerate a brief window when the ledger is down, and resume operation as soon as the new ledger ID is delivered.

Note that this service will be available on the sandbox and might be available in some other testing environments, but will *never* be available in production.

Method name	Request type	Response type	Description
Reset	*ResetRequest*	.google.protobuf.Empty	Resets the ledger state. Note that loaded DARs won't be removed – this only rolls back the ledger to genesis.

3.8.3.17 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`). Required

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`). Required
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)

3.8.3.18 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
created	*CreatedEvent*		
exercised	*ExercisedEvent*		

3.8.3.19 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*		

3.8.3.20 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`). Required

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`). Required
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`). Required
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`). Required
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 or if the ledger has been pruned before `begin` - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields (e.g. if `before` is not before `end`) - `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 or if the ledger has been pruned before `begin` - `INVALID_ARGUMENT`: if the payload is malformed or is missing required fields (e.g. if `before` is not before `end`) - `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)
GetFlatTransactionByEventId	*GetFlatTransactionByEventIdRequest*	*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 - `PERMISSION_DENIED`: if the claims in the

3.8.3.21 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 strings as identifiers. Those classes are defined as follow: - NameStrings are strings that match the regexp `[A-Za-z\$_][A-Za-z0-9\$_]*`. - PackageIdStrings are strings that match the regexp `[A-Za-z0-9\-_ ]+`. - PartyIdStrings are strings that match the regexp `[A-Za-z0-9:\-_ ]+`. - LedgerStrings are strings that match the regexp `[A-Za-z0-9#:\-_/ ]+`.

Field	Type	Label	Description
record	*Record*		
variant	*Variant*		
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.
list	*List*		Represents a homogeneous list of values.
int64	*sint64*		
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.
text	*string*		A string.
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
party	*string*		An agent operating on the ledger. Must be a valid PartyIdString.
bool	*bool*		True or false.
unit	google.protobuf.Empty		This value is used for example for choices that don't take any arguments.
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
optional	*Optional*		The Optional type, None or Some
map	*Map*		The Map type
enum	*Enum*		The Enum type
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

3.8.3.22 com/daml/ledger/api/v1/version_service.proto

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`). Required

GetLedgerApiVersionResponse

Field	Type	Label	Description
version	*string*		The version of the ledger API

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

3.8.3.23 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

3.8.4 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*.

3.8.4.1 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.

3.8.4.2 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*:

```
    alice <- getParty "Alice"
    someCid <- submit alice do create SomeTemplate with owner=alice
    let myProduct = MyProductType with
                intField = 17
                textField = "some text"
                decimalField = 17.42
```

(continues on next page)

(continued from previous page)

```
                boolField = False
                partyField = bob
                timeField = datetime 2018 May 16 0 0 0
                listField = [1,2,3]
                contractIdField = someCid
```

For this data, the respective data on the Ledger API is shown below. Note that this value would be enclosed by a particular contract containing a field of type *MyProductType*. See *Contract templates* for the translation of Daml contracts to the representation by the Ledger API.

```
{ // Record
  record_id { // Identifier
    package_id: "some-hash"
    name: "Types.MyProductType"
  }
  fields { // RecordField
    label: "intField"
    value { // Value
      int64: 17
    }
  }
  fields { // RecordField
    label: "textField"
    value { // Value
      text: "some text"
    }
  }
  fields { // RecordField
    label: "decimalField"
    value { // Value
      decimal: "17.42"
    }
  }
  fields { // RecordField
    label: "boolField"
    value { // Value
      bool: false
    }
  }
  fields { // RecordField
    label: "partyField"
    value { // Value
      party: "Bob"
    }
  }
  fields { // RecordField
    label: "timeField"
    value { // Value
      timestamp: 1526428800000000
```

(continues on next page)

(continued from previous page)

```
    }
  }
  fields { // RecordField
    label: "listField"
    value { // Value
      list { // List
        elements { // Value
          int64: 1
        }
        elements { // Value
          int64: 2
        }
        elements { // Value
          int64: 3
        }
      }
    }
  }
  fields { // RecordField
    label: "contractIdField"
    value { // Value
      contract_id: "some-contract-id"
    }
  }
}
```

3.8.4.3 Variants

Variants or sum types are types with multiple constructors. This example defines a simple variant type with two constructors:

```
data MySumType = MySumConstructor1 Int |
                 MySumConstructor2 (Text, Bool)
```

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

```
    let mySum1 = MySumConstructor1 17
    let mySum2 = MySumConstructor2 ("it's a sum", True)
```

Similar to records, variants are also enclosed by a contract, a record, or another variant.

The snippets below shows the value of `mySum1` and `mySum2` respectively as they would be transmitted on the Ledger API within a contract.

Listing 1: mySum1

```
{ // Value
  variant { // Variant
    variant_id { // Identifier
```

(continues on next page)

(continued from previous page)

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

Listing 2: 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
          }
        }
      }
    }
  }
}
```

3.8.4.4 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
```

(continues on next page)

(continued from previous page)

```
    with
        owner: Party
    where
        signatory owner

        key MySimpleTemplateKey owner: MySimpleTemplateKey
```

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"
```

(continues on next page)

(continued from previous page)

```
}
```

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
```

(continues on next page)

(continued from previous page)

```
      fields { // RecordField
        label: "parameter"
        value { // Value
          int64: 42
        }
      }
    }
  }
}
```

3.8.5 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*.

3.8.5.1 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`.

3.8.5.2 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`.

3.8.5.3 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 }`

3.8.5.4 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 }
```

3.8.5.5 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:

```
    controller owner can
      nonconsuming DoNothing: ()
        do
          return ()

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

3.8.5.6 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`

3.8.6 Java bindings

3.8.6.1 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 3: Daml

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

Listing 4: 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 5: 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 6: 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 7: com/acme/producttypes.Name.java

```
package com.acme.producttypes;

public class Name {
  public final String firstName;
  public final String lastName;

  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. .. TODO: refer to another section explaining exactly that, when we have it.

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 8: 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
```

(continues on next page)

(continued from previous page)

```
    controller owner can
      Bar_SomeChoice: Bool
        with
        aName: Text
          do return True
```

A file is generated that defines three Java classes:

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

Listing 9: 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;
```

(continues on next page)

(continued from previous page)

```
    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 10: 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 11: 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 12: 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 13: 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 14: 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 15: 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 16: 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 17: 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 18: Com/Acme/Enum.daml

```
module Com.Acme.Enum where

data Color = Red | Blue | Green
```

The Java code generated for this variant is:

Listing 19: com/acme/enum/Color.java

```
package com.acme.enum;


public enum Color {
```

(continues on next page)

(continued from previous page)

```
  RED,

  GREEN,

  BLUE;

  /* ... */

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

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

Listing 20: com/acme/enum/bookattribute/Authors.java

```
package com.acme.enum.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() { /* ... */ }

}
```

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"))
```

(continues on next page)

(continued from previous page)

```
            .toList(
                f1 -> f1.asText().orElseThrow(() -> new
↪IllegalArgumentException("Expected Text element"))
                        .getValue()
            )
    );
```

3.8.6.2 Example project

To try out the Java bindings library, use the examples on GitHub: `PingPongReactive` or `PingPongComponents`.

The former example does not use the Reactive Components, and the latter example does. Both examples implement 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 three examples of the PingPong application, built with gRPC (non-reactive), Reactive and Reactive Component bindings respectively.

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.

3.8.6.3 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 scenarios* 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 .daml/dist/quickstart-0.0.1.dar`
   The output should look like this:

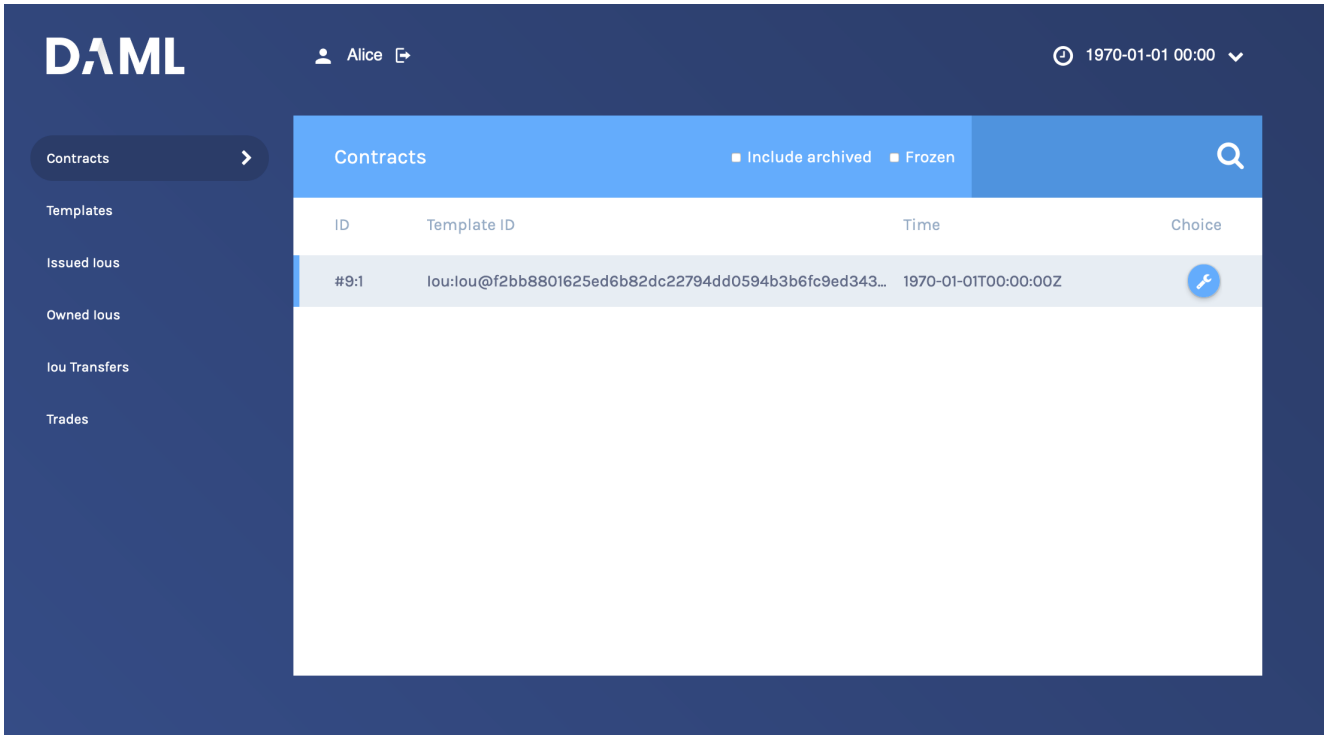
```
INFO: Slf4jLogger started
INFO: Listening on localhost:6865 over plain text.

   ____             ____
  / __/__ ____  ___/ / /  ___ __ __
 _\ \/ _ `/ _ \/ _  / _ \/ _ \\ \ /
/___/\_,_/_//_/\_,_/_.__/\___/_\_\

INFO: Initialized sandbox version 1.6.0 with ledger-id = 3528b0dd-069b-
↪420c-9df4-4af6b8fd4f47, port = 6865, dar file = List(.daml/dist/
↪quickstart-0.0.1.dar), time mode = wall-clock time, ledger = in-
↪memory, auth-service = AuthServiceWildcard$, contract ids seeding =
↪strong
```

   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:

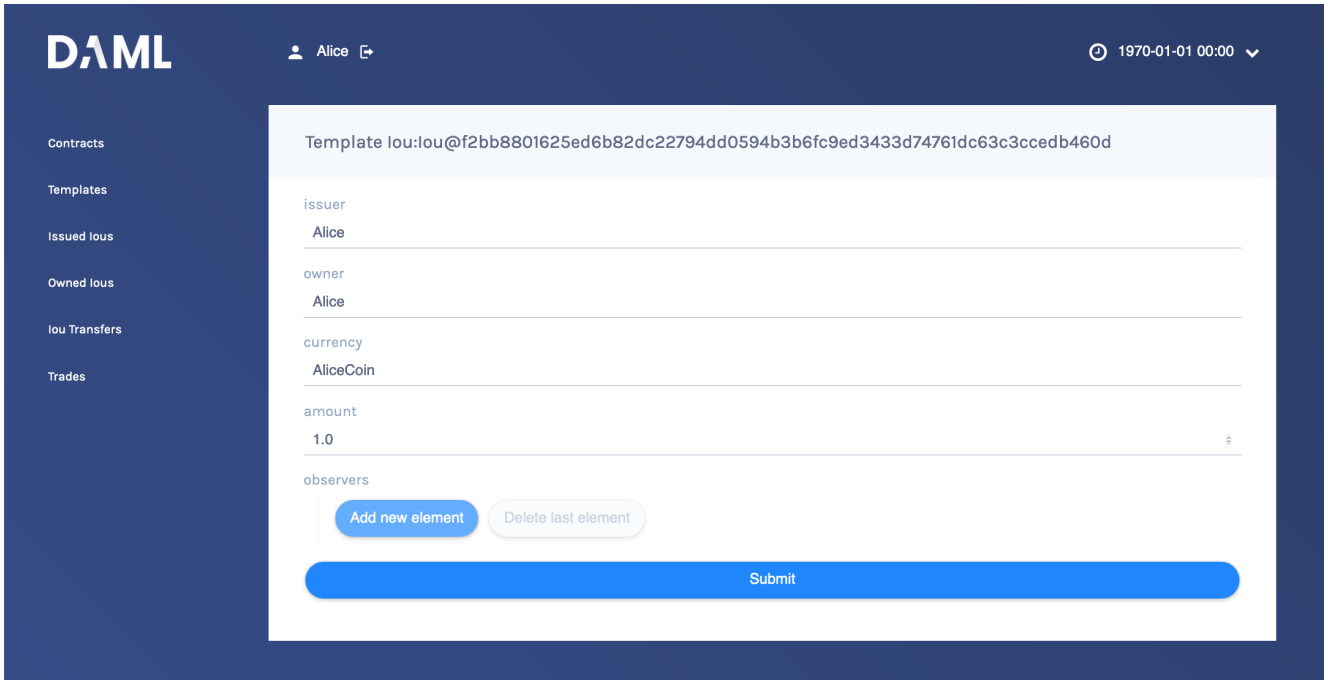
1. Go to http://localhost:4000/. This is the Navigator, which you launched *earlier*.
2. On the login screen, select **Alice** from the dropdown. This logs you in as Alice.
   (The list of available parties is specified in the `ui-backend.conf` file.)
   This takes you to the contracts view:
   This 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 walkthrough, 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.

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

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*.

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 scenarios* 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 two lines specify language version and 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, *rights* or *choices* are given to `owner`:

```
controller owner can
  Iou_Transfer : ContractId IouTransfer
    with
      newOwner : Party
    do create IouTransfer with iou = this; newOwner
```

`controller owner can` starts the block. In this case, `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`.

```
controller seller can
  IouTrade_Accept : (IouCid, IouCid)
    with
      quoteIouCid : IouCid
    do
      baseIou <- fetch baseIouCid
      baseIssuer === baseIou.issuer
      baseCurrency === baseIou.currency
      baseAmount === baseIou.amount
      buyer === baseIou.owner
      quoteIou <- fetch quoteIouCid
      quoteIssuer === quoteIou.issuer
```

(continues on next page)

(continued from previous page)

```
          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 scenarios

You can check the correct authorization and privacy of a contract model using *scenarios*: tests that are written in Daml.

Scenarios 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 scenarios in the quickstart application, open `daml/Tests/Trade.daml` in Daml Studio.

A scenario test is defined with `trade_test = scenario 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
```

Compare the scenario with the `setup` scenario in `daml/Main.daml`. You will see that the scenario you used to initialize the sandbox is an initial segment of the `trade_test` scenario. 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 *Scenario 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:61:14)
#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)
    └─> fetch #4:1 (Iou:Iou)

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

    #6:3
    │   known to (since): 'Bob' (#6), 'USD_Bank' (#6), 'Alice' (#6)
    └─> 'Bob' exercises Iou_Transfer on #3:1 (Iou:Iou)
              with
                newOwner = 'Alice'
```

(continues on next page)

(continued from previous page)

```
        children:
        #6:4
        │   consumed by: #6:5
        │   referenced by #6:5
        │   known to (since): 'Bob' (#6), 'USD_Bank' (#6), 'Alice' (#6)
        └─> create Iou:IouTransfer
            with
              iou =
                (Iou:Iou with
                   issuer = 'USD_Bank';
                   owner = 'Bob';
                   currency = "USD";
                   amount = 110.0;
                   observers = []);
              newOwner = 'Alice'

    #6:5
    │   known to (since): 'Bob' (#6), 'USD_Bank' (#6), 'Alice' (#6)
    └─> 'Alice' exercises IouTransfer_Accept on #6:4 (Iou:IouTransfer)
                with
        children:
        #6:6
        │   referenced by #7:0
        │   known to (since): 'Alice' (#6), 'USD_Bank' (#6), 'Bob' (#6)
        └─> create Iou:Iou
            with
              issuer = 'USD_Bank';
              owner = 'Alice';
              currency = "USD";
              amount = 110.0;
              observers = []

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

(continues on next page)

(continued from previous page)

```
                    amount = 100.0;
                    observers = ['Bob']);
                newOwner = 'Bob'

    #6:9
    │   known to (since): 'Alice' (#6), 'EUR_Bank' (#6), 'Bob' (#6)
    └─> 'Bob' exercises IouTransfer_Accept on #6:8 (Iou:IouTransfer)
                with
        children:
        #6:10
        │   referenced by #8:0
        │   known to (since): 'Bob' (#6), 'EUR_Bank' (#6), 'Alice' (#6)
        └─> create Iou:Iou
            with
              issuer = 'EUR_Bank'; owner = 'Bob'; currency = "EUR"; amount
↪= 100.0; observers = []
```

The `submit` function used in this scenario 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 Connect 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();

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

5. Commands are submitted via the Command Submission Service.

```
  return client
```

(continues on next page)

(continued from previous page)

```
        .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 21: ExerciseCommand

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

Listing 22: 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.

3.8.6.4 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.

3.8.6.5 Reference documentation

Click here for the JavaDoc reference documentation.

3.8.6.6 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`.

3.8.6.7 Example project

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

3.8.7 Scala bindings

The Scala bindings are deprecated as of 2020-10-14.

This page provides a basic Scala programmer's introduction to working with Daml Ledgers, using the Scala programming language and the **Ledger API**.

3.8.7.1 Introduction

The Scala bindings is a client implementation of the **Ledger API**. The Scala bindings library lets you write applications that connect to a Daml Ledger using the Scala programming language.

There are two main components:

Scala codegen Daml to Scala code generator. Use this to generate Scala classes from Daml models. The generated Scala code provides a type safe way of creating contracts (*CreateCommand*) and exercising contract choices (*ExerciseCommand*).

Akka Streams-based API The API that you use to send commands to the ledger and receive transactions back.

In order to use the Scala bindings, you should be familiar with:

Daml language
Ledger API
Akka Streams API
Scala programming language
Building Daml projects
Daml codegen

3.8.7.2 Getting started

If this is your first experience with the Scala bindings library, we recommend that you start by looking at the quickstart-scala example.

To use the Scala bindings, set up the following dependencies in your project:

```
lazy val codeGenDependencies = Seq(
  "com.daml" %% "bindings-scala" % damlSdkVersion
)

lazy val applicationDependencies = Seq(
  "com.daml" %% "bindings-akka" % damlSdkVersion
)
```

We recommend separating generated code and application code into different modules. There are two modules in the `quickstart-scala` example:

`scala-codegen` This module will contain only generated Scala classes.
`application` This is the application code that makes use of the generated Scala classes.

```
lazy val `scala-codegen` = project
  .in(file("scala-codegen"))
  .settings(
    name := "scala-codegen",
    commonSettings,
    libraryDependencies ++= codeGenDependencies,
  )

lazy val `application` = project
  .in(file("application"))
  .settings(
    name := "application",
    commonSettings,
    libraryDependencies ++= codeGenDependencies ++ applicationDependencies,
  )
  .dependsOn(`scala-codegen`)
```

3.8.7.3 Generating Scala code

1) Install *the latest version of the SDK*.
2) Build a **DAR** file from a **Daml** model. Refer to *Building Daml projects* for more instructions.
3) Configure `codegen` in the `daml.yaml` (for more details see *Daml codegen* documentation).

```
codegen:
  scala:
    package-prefix: com.daml.quickstart.iou.model
    output-directory: scala-codegen/src/main/scala
    verbosity: 2
```

4) Run Scala codegen:

```
$ daml codegen scala
```

If the command is successful, it should print:

```
Scala codegen
Reading configuration from project configuration file
[INFO ] Scala Codegen verbosity: INFO
[INFO ] decoding archive with Package ID:
↪5c96aa21d5f38386833ff47fe1a7562afb5b3fe5be520f289c42892dfb0ef42b
[INFO ] decoding archive with Package ID:
↪748d55be531976e941076a44fe8c06ad4a7bdb36160711dd0204b5ab8dc77e44
[INFO ] decoding archive with Package ID:
↪d841a5e45897aea965ab7699f3e51613c9d00b9fbd1bb09658d7fb00486f5b57
[INFO ] Scala Codegen result:
Number of generated templates: 3
Number of not generated templates: 0
Details:
```

The output above tells that Scala codegen read configuration from `daml.yaml` and produced Scala classes for 3 templates without errors (empty `Details:` line).

3.8.7.4 Example code

In this section we will demonstrate how to use the Scala bindings library.

This section refers to the IOU Daml example from the *Quickstart guide* and quickstart-scala example that we already mentioned above.

Please keep in mind that **quickstart-scala example** compiles with `-Xsource:2.13` **scalac** option, this is to activate the fix for a Scala bug that forced users to add extra imports for implicits that should not be needed.

Create a contract and send a CreateCommand

To create a Scala class representing an **IOU** contract, you need the following **imports**:

```
import com.daml.ledger.client.binding.{Primitive => P}
import com.daml.quickstart.iou.model.{Iou => M}
```

the definition of the **issuer** `Party`:

```
  private val issuer = P.Party("Alice")
```

and the following code to create an instance of the `M.Iou` class:

```
  val iou = M.Iou(
    issuer = issuer,
    owner = issuer,
    currency = "USD",
    amount = BigDecimal("1000.00"),
    observers = List(),
  )
```

To send a *CreateCommand* (keep in mind the following code snippet is part of the Scala *for comprehension expression*):

```
    createCmd = iou.create
    _ <- clientUtil.submitCommand(issuer, issuerWorkflowId, createCmd)
    _ = logger.info(s"$issuer created IOU: $iou")
    _ = logger.info(s"$issuer sent create command: $createCmd")
```

For more details on how to submit a command, please refer to the implementation of com.daml.quickstart.iou.ClientUtil#submitCommand.

Receive a transaction, exercise a choice and send an ExerciseCommand

To receive a transaction as a **newOwner** and decode a *CreatedEvent* for `IouTransfer` contract, you need the definition of the **newOwner** `Party`:

```
  private val newOwner = P.Party("Bob")
```

and the following code that handles subscription and decoding:

```
    _ <- clientUtil.subscribe(newOwner, offset0, None) { tx =>
      logger.info(s"$newOwner received transaction: $tx")
      decodeCreated[M.IouTransfer](tx).foreach { contract: Contract[M.
↪IouTransfer] =>
        logger.info(s"$newOwner received contract: $contract")
```

To exercise `IouTransfer_Accept` choice on the `IouTransfer` contract that you received and send a corresponding *ExerciseCommand*:

```
        val exerciseCmd = contract.contractId.exerciseIouTransfer_
↪Accept(actor = newOwner)
        clientUtil.submitCommand(newOwner, newOwnerWorkflowId,
↪exerciseCmd) onComplete {
          case Success(_) =>
            logger.info(s"$newOwner sent exercise command: $exerciseCmd")
            logger.info(s"$newOwner accepted IOU Transfer: $contract")
          case Failure(e) =>
            logger.error(s"$newOwner failed to send exercise command:
↪$exerciseCmd", e)
        }
```

Fore more details on how to subscribe to receive events for a particular party, please refer to the implementation of com.daml.quickstart.iou.IouMain#newOwnerAcceptsAllTransfers.

3.8.7.5 Authorization

Some ledgers will require you to send an access token along with each request. To learn more about authorization, read the *Authorization* overview.

To use the same token for all ledger API requests, use the `token` field of `LedgerClientConfiguration`:

```
  private val clientConfig = LedgerClientConfiguration(
    applicationId = ApplicationId.unwrap(applicationId),
    ledgerIdRequirement = LedgerIdRequirement.none,
    commandClient = CommandClientConfiguration.default,
    sslContext = None,
    token = None,
  )
```

To specify the token for an individual call, use the `token` parameter:

```
transactionClient.getLedgerEnd() // Uses the token specified in
↪LedgerClientConfiguration
transactionClient.getLedgerEnd(token = accessToken) // Uses the given token
```

Note that if your tokens can change at run time (e.g., because they expire or because you switch users), you will need to specify them on a per-call basis as shown above.

3.8.8 Node.js bindings

The Node.js bindings are deprecated as of 2020-10-14.

The documentation for the Node.js bindings has been moved to digital-asset.github.io/daml-js.

You can also try the Node.js bindings tutorial, which is at github.com/digital-asset/ex-tutorial-nodejs.

3.8.9 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

Scala macro annotations (not future-proof enough to use when implementing the last Scala codegen)

3.8.9.1 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 generated by Scala codegen.

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]
```

Here is how to manually build a *CreateCommand* for the above contract template in Scala:

```
  def iouCreateCommand(
      templateId: Identifier,
      issuer: String,
      owner: String,
      currency: String,
      amount: BigDecimal,
  ): Command.Create = {
    val fields = Seq(
      RecordField("issuer", Some(Value(Value.Sum.Party(issuer)))),
      RecordField("owner", Some(Value(Value.Sum.Party(owner)))),
      RecordField("currency", Some(Value(Value.Sum.Text(currency)))),
      RecordField("amount", Some(Value(Value.Sum.Numeric(amount.
↪toString)))),
      RecordField("observers", Some(Value(Value.Sum.List(List()))))),
    )
    Command.Create(
      CreateCommand(
        templateId = Some(templateId),
        createArguments = Some(Record(Some(templateId), fields)),
      )
    )
  }
```

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.

Codegen should simplify the command construction by providing auto-generated utilities to help you construct commands. For example, when you use *Scala codegen* to generate contract classes, a

similar contract instantiation would look like this:

```
val iou = M.Iou(
  issuer = issuer,
  owner = issuer,
  currency = "USD",
  amount = BigDecimal("1000.00"),
  observers = List(),
)
```

Exercise Command

To build *ExerciseCommand* for *Iou_Transfer*:

```
  controller owner can
    Iou_Transfer : ContractId IouTransfer
      with
        newOwner : Party
      do create IouTransfer with iou = this; newOwner
```

manually in Scala:

```
def iouTransferExerciseCommand(
    templateId: Identifier,
    contractId: String,
    newOwner: String,
): Command.Exercise = {
  val transferTemplateId = Identifier(
    packageId = templateId.packageId,
    moduleName = templateId.moduleName,
    entityName = "Iou_Transfer",
  )
  val fields = Seq(RecordField("newOwner", Some(Value(Value.Sum.
↪Party(newOwner)))))
  Command.Exercise(
    ExerciseCommand(
      templateId = Some(templateId),
      contractId = contractId,
      choice = "Iou_Transfer",
      choiceArgument = Some(Value(Value.Sum.
↪Record(Record(Some(transferTemplateId), fields)))),
    )
  )
}
```

versus creating the same command using a value class generated by *Scala codegen*:

```
  exerciseCmd = iouContract.contractId.exerciseIou_Transfer(actor =
↪issuer, newOwner = newOwner)
```

3.8.9.2 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.

3.8.9.3 Links

A Scala example that demonstrates how to manually construct ledger commands: https://github.com/digital-asset/daml/tree/main/language-support/scala/examples/iou-no-codegen
A Scala codegen example: https://github.com/digital-asset/daml/tree/main/language-support/scala/examples/quickstart-scala
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.

3.8.10 What's in the Ledger API

You can access the Ledger API via the HTTP JSON API, Java bindings, Scala 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 time for commands.
- Use the *version service* to retrieve information about the Ledger API version.

Testing services (on Sandbox only, *not* for production ledgers)

- Use the *time service* to obtain the time as known by the ledger.
- Use the *reset service* to reset the ledger state, as a quicker alternative to restarting the whole ledger application.

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.

3.8.11 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)

3.8.11.1 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.

Chapter 4

Deploying to Daml ledgers

4.1 Overview of Daml ledgers

This is an overview of Daml deployment options. Instructions on how to deploy to a specific ledger are available in the following section.

4.1.1 Commercial Integrations

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

Product	Ledger	Vendor
Daml on Corda	Corda	Multiple. Contact Digital Asset
Sextant for Daml	Amazon Aurora	Blockchain Technology Partners
Sextant for Daml	Hyperledger Sawtooth	Blockchain Technology Partners
Sextant for Daml	Amazon QLDB	Blockchain Technology Partners
Sextant for Daml	Hyperledger Besu	Blockchain Technology Partners
Daml Hub	Managed cloud enviroment	Digital Asset

4.1.2 Open Source Integrations

The following table lists open source Daml integrations.

Ledger	Developer	More Information
Hyperledger Sawtooth	Blockchain Technology Partners	Github Repo
Hyperledger Fabric	Digital Asset	Github Repo
PostgreSQL	Digital Asset	Daml Driver for PostgreSQL Docs

4.1.3 Daml Ledgers in Development

The following table lists the ledgers that are implementing support for running Daml.

Ledger	Developer	More Information
VMware Blockchain	VMware	Press release, April 2019
FISCO BCOS	WeBank	Press release, April 2020
Canton	Digital Asset reference implementation	canton.io

4.2 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:

```yaml
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 more further 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 `ledger.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
  navigator                Launch Navigator on ledger
```

4.2.1 Connecting via TLS

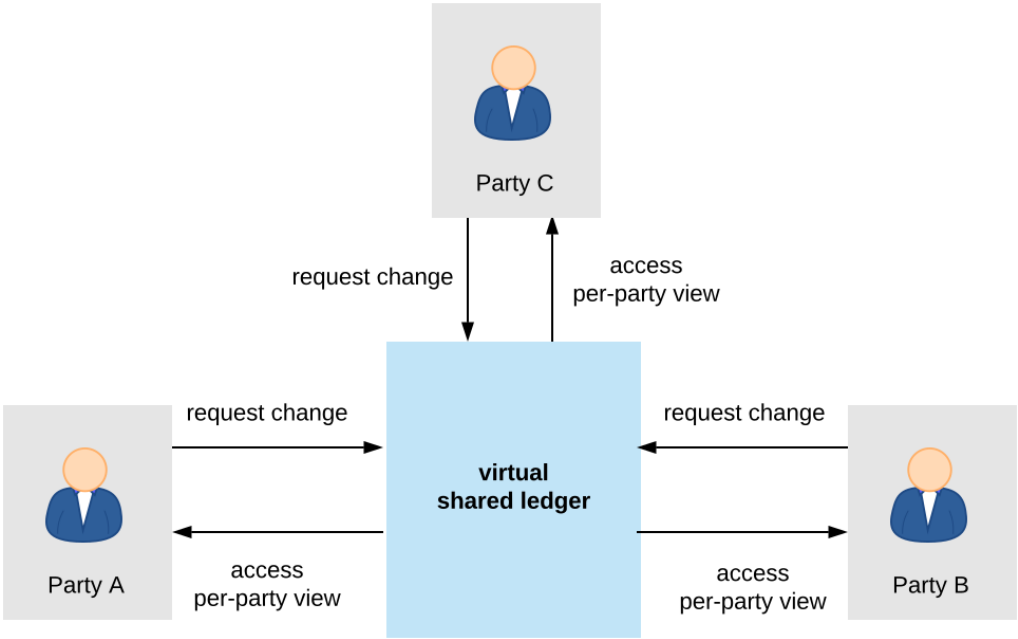
To connect to the ledger via TLS, you can pass `--tls` to the various commands. If your ledger supports or requires mutual authentication you can pass your client key and certificate chain files via `--pem client_key.pem --crt client.crt`. Finally, you can use a custom certificate authority for validating the server certificate by passing `--cacrt server.crt`. If `--pem`, `--crt` or `--cacrt` are specified TLS is enabled automatically so `--tls` is redundant.

4.2.2 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.

4.3 Daml Ledger Topologies

The Ledger API provides parties with an abstraction of a virtual shared ledger, visualized as follows.



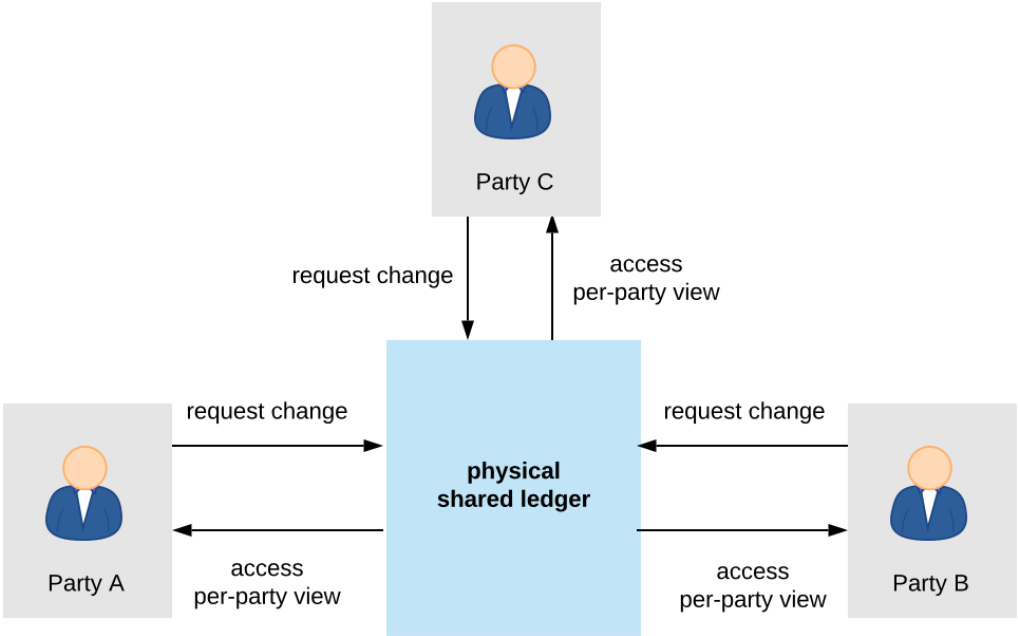
The real-world topologies of actual ledger implementations differ significantly, however. The topologies can impact both the functional and non-functional properties of the resulting ledger. This document provides one useful categorization of the existing implementations' topologies: the split into global and partial state topologies, depending on whether single *trust domains* can see the entire ledger, or just parts of it. The implementations with topologies from the same category share many non-functional properties and trust assumptions. Additionally, their *identity and package management* functions also behave similarly.

4.3.1 Global State Topologies

In global state topologies, there exists at least one *trust domain* whose systems contain a physical copy of the entire virtual shared ledger that is accessible through the API.

4.3.1.1 The Fully Centralized Ledger

The simplest global state topology is the one where the virtual shared ledger is implemented through a single machine containing a physical copy of the shared ledger, whose real-world owner is called the **operator**.



The *Daml Sandbox* uses this topology. While simple to deploy and operate, the single-machine setup also has downsides:

1. it provides no scaling
2. it is not highly available
3. the operator is fully trusted with preserving the ledger's integrity
4. the operator has full insight into the entire ledger, and is thus fully trusted with privacy
5. it provides no built-in way to interoperate (transactionally share data) across several deployed ledgers; each deployment defines its own segregated virtual shared ledger.

The first four problems can be solved or mitigated as follows:

1. scaling by splitting the system up into separate functional components and parallelizing execution
2. availability by replication
3. trust for integrity by introducing multiple trust domains and distributing trust using Byzantine fault tolerant replication, or by maintaining one trust domain but using hardware-based Trusted Execution Environments (TEEs) or other cryptographic means to enforce or audit ledger integrity without having to trust the operator.
4. trust for privacy through TEEs that restrict data access by hardware means.

The remainder of the section discusses these solutions and their implementations in the different Daml ledgers. The last problem, interoperability, is inherent when the two deployments are operated by different trust domains: by definition, a topology in which no single trust domain would hold the entire ledger is not a global state topology.

4.3.1.2 Scaling

The main functionalities of a system providing the Ledger API are:

1. serving the API itself: handling the gRPC connections and authorizing users,
2. allowing the API users to access their *ledger projection* (reading the ledger), and
3. allowing the API users to issue commands and thus attempt to append commits to the shared ledger (writing to the ledger).

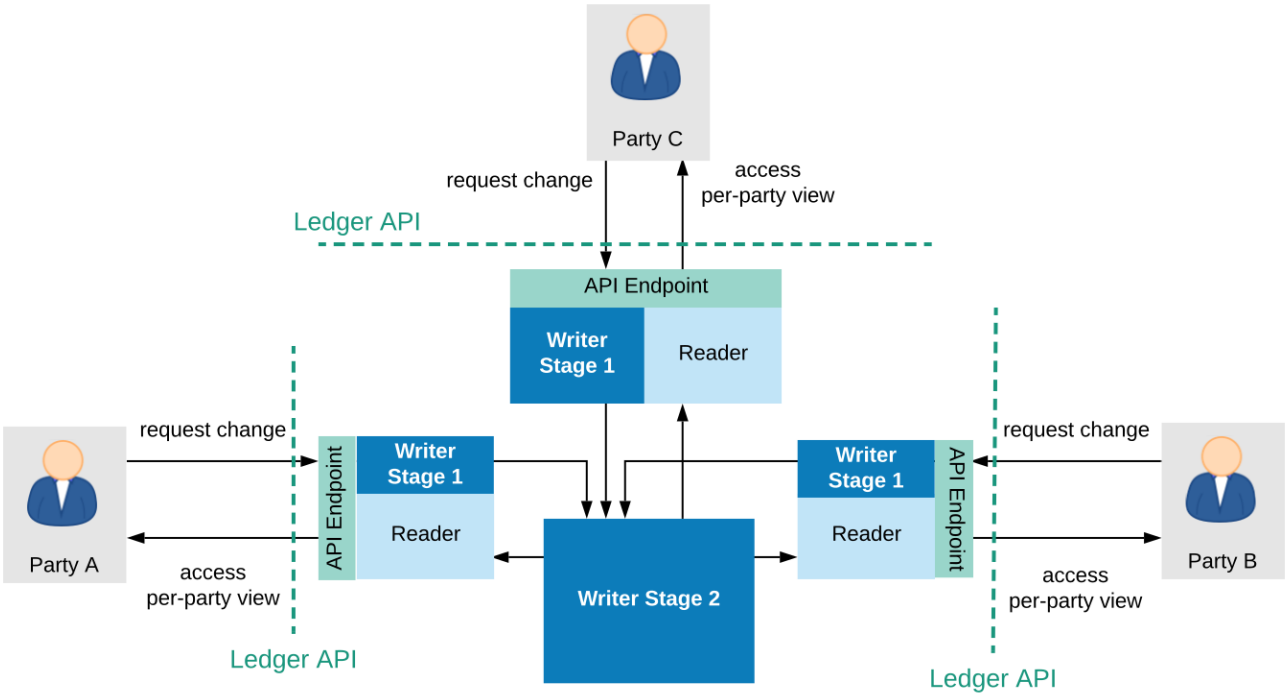
The implementation thus naturally splits up into components for serving the API, reading from the ledger, and writing to the ledger. Serving the API and reading can be scaled out horizontally. Reading can be scaled out by building caches of the ledger contents; as the projections are streams, no synchronization between the different caches is necessary.

To ensure ledger integrity, the writing component must preserve the ledger's *validity conditions*. Writing can thus be further split up into three sub-components, one for each of the three validity conditions:

1. *model conformance* checks (i.e., Daml interpretation),
2. *authorization* checks, and
3. *consistency* checks.

Of these three, conformance and authorization checks can be checked in isolation for each commit. Thus, such checks can be parallelized and scaled out. The consistency check cannot be done in isolation and requires synchronization. However, to improve scaling, it can internally still use some form of sharding, together with a commit protocol.

For example, the next versions of Daml on Amazon Aurora and on Hyperledger Fabric will use such partitioned topologies. The next image shows an extreme version of this partitioning, where each party is served by a separate system node running all the parallelizable functions. The writing subsystem is split into two stages. The first stage checks conformance and authorization, and can be arbitrarily replicated, while the second stage is centralized and checks consistency.



4.3.1.3 Replication: Availability and Distributing Trust

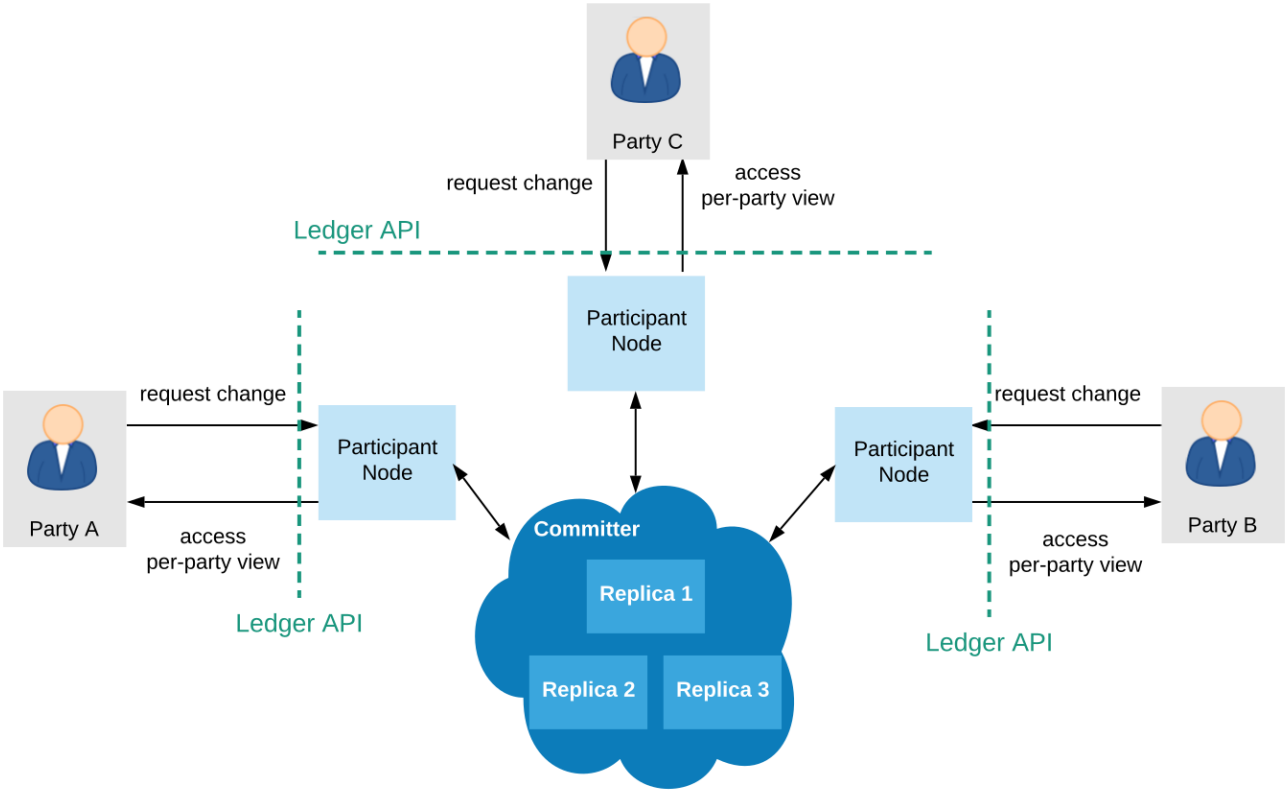
Availability is improved by replication. The scaling methodology described in the previous section already improves the ledger's availability properties, as it introduces replication for most functions.

For example, if a node serving a client with the API fails, clients can fail over to other such nodes. Replicating the writer's consistency-checking subsystem must use a consensus algorithm to ensure consistency of the replicated system (in particular, the linearizability of the virtual shared ledger).

Replication can also help to lower, or more precisely distribute the trust required to ensure the system's integrity. Trust can be distributed by introducing multiple organizations, i.e., multiple trust domains into the system. In these situations, the system typically consists of two types of nodes:

1. **Writer nodes**, which replicate the physical shared ledger and can extend it with new commits. Writer nodes are thus also referred to as **committer nodes**.
2. **Participant nodes**, (also called Client nodes in some platforms) which serve the Ledger API to a subset of the system parties, which we say are hosted by this participant. A participant node proposes new commits on behalf of the parties it hosts, and holds a portion of the ledger that is relevant for those parties (i.e., the parties' *ledger projection*). The term participant node is sometimes also used more generally, for any physical node serving the Ledger API to a party.

The participant nodes need not be trusted by the other nodes, or by the committer(s); the participants can be operated by mutually distrusting entities, i.e., belong to different trust domains. In general, the participant nodes do not necessarily even need to know each other. However, they have to be known to and accepted by the committer nodes. The committer nodes are jointly trusted with ensuring the ledger's integrity. To distribute the trust, the committer nodes must implement a Byzantine fault tolerant replication mechanism. For example, the mechanism can ensure that the system preserves integrity even if up to a third of the committer nodes (e.g., 2 out of 7) misbehave in arbitrary ways. The resulting topology is visualized below.



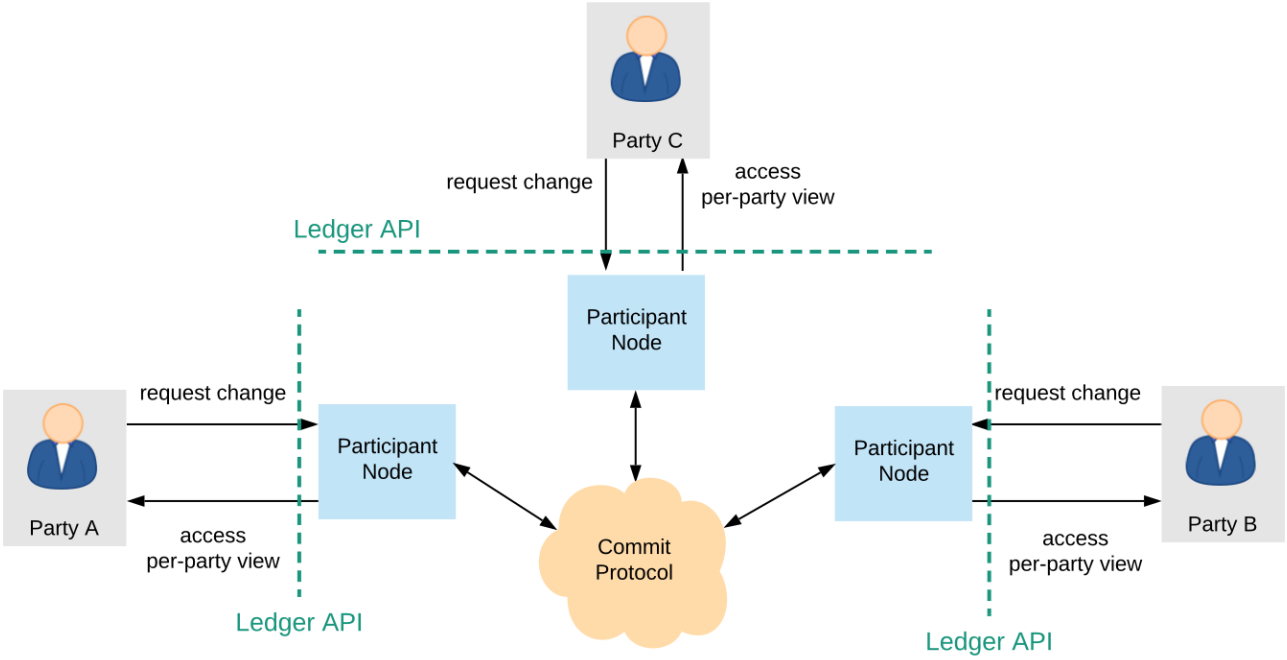
Daml on VMware Concord and Daml on Hyperledger Sawtooth are examples of such a replicated setup.

4.3.1.4 Trusted Execution Environments

Integrity and privacy can also be protected using hardware Trusted Execution Environments (TEEs), such as Intel SGX. The software implementing the ledger can then be deployed inside of TEE **enclaves**, which are code blocks that the processor isolates and protects from the rest of the software stack (even the operating system). The hardware ensures that the enclave data never leaves the processor unencrypted, offering privacy. Furthermore, hardware-based attestation can guarantee that the operating entities process data using the prescribed code only, guaranteeing integrity. The hardware is designed in such a way as to make any potential physical attacks by the operator extremely expensive. This moves the trust necessary to achieve these properties from the operators of the trust domains that maintain the global state to the hardware manufacturer, who is anyway trusted with correctly producing the hardware. Recent security research has, however, found scenarios where the TEE protection mechanisms can be compromised.

4.3.2 Partitioned Ledger Topologies

In these topologies, the ledger is implemented as a distributed system. Unlike the global state topologies, no single trust domain holds a physical copy of the entire shared ledger. Instead, the participant nodes hold just the part of the ledger (i.e., the *ledger projection*) that is relevant to the parties to whom they serve the Ledger API. The participants jointly extend the ledger by running a distributed commit protocol.



The implementations might still rely on trusted third parties to facilitate the commit protocol. The required trust in terms of privacy and integrity, however, can generally be lower than in global state topologies. Moreover, unlike the previous topologies, they support interoperability: even if two transactions are committed with the help of disjoint sets of trusted third parties, their *output contracts* can in general still be used within the same atomic transaction. The exact trust assumptions and the degree of supported interoperability are implementation-dependent. Canton and Daml on R3 Corda are two such implementations. The main drawback of this topology is that availability can be influenced by the participant nodes. In particular, transactions cannot be committed if they use data that is only stored on unresponsive nodes. Spreading the data among additional trusted entities can mitigate the problem.

Chapter 5

Operating Daml

5.1 Daml 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.

5.1.1 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.
> 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

[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.

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.

5.1.2 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 `NOT_FOUND` 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.

5.1.3 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.

5.1.4 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.

[3] This might be improved in future versions.

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.

5.1.5 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.

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

Chapter 6

Developer Tools

6.1 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 Connect 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 *Extractor*: `daml extractor`
- 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.

6.1.1 Full help for commands

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

6.1.2 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

6.1.2.1 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 Connect 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
```

6.1.2.2 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__
platform-version: __VERSION__
name: __PROJECT_NAME__
source: daml
scenario: Main:setup
parties:
  - Alice
  - Bob
version: 1.0.0
exposed-modules:
  - Main
```

(continues on next page)

(continued from previous page)

```
dependencies:
  - daml-prim
  - daml-stdlib
scenario-service:
  grpc-max-message-size: 134217728
  grpc-timeout: 60
  jvm-options: []
build-options: ["--ghc-option", "-Werror",
                "--ghc-option", "-v"]
```

Here is what each field means:

`sdk-version`: the SDK version that this project uses.

The assistant automatically downloads and installs this version if needed (see the `auto-install` setting in the global config). We recommend keeping this up to date with the latest stable release of the SDK. It is possible to override the version without modifying the `daml.yaml` file by setting the `DAML_SDK_VERSION` environment variable. This is mainly useful when you are working with an external project that you want to build with a specific version.

The assistant will warn you when it is time to update this setting (see the `update-check` setting in the global config to control how often it checks, or to disable this check entirely).

`platform-version`: Optional SDK version of platform components. Not setting this is equivalent to setting it to the same version as `sdk-version`. At the moment this includes Sandbox, Sandbox classic and the HTTP JSON API both when invoked directly via `daml sandbox` as well as when invoked via `daml start`. Changing the platform version is useful if you deploy to a ledger that is running on a different SDK version than you use locally and you want to make sure that you catch any issues during testing. E.g., you might compile your Daml code using SDK 1.3.0 so you get improvements in Daml Studio but deploy to Daml Hub which could still be running a ledger and the JSON API from SDK 1.2.0. In that case, you can set `sdk-version: 1.3.0` and `platform-version: 1.2.0`. It is possible to override the platform version by setting the `DAML_PLATFORM_VERSION` environment variable.

`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.

`scenario`: the name of the scenario to run when using `daml start`.

`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*.

`scenario-service`: settings for the scenario 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 scenario 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 scenario 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`.

6.1.2.3 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.

6.1.3 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
```

6.1.4 Managing releases

You can manage SDK versions manually by using `daml install`.

To download and install SDK of the latest stable Daml Connect 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 `1.7.0`, run:

```
daml install 1.7.0
```

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.

6.1.5 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`.

6.1.6 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.

6.2 Daml Studio

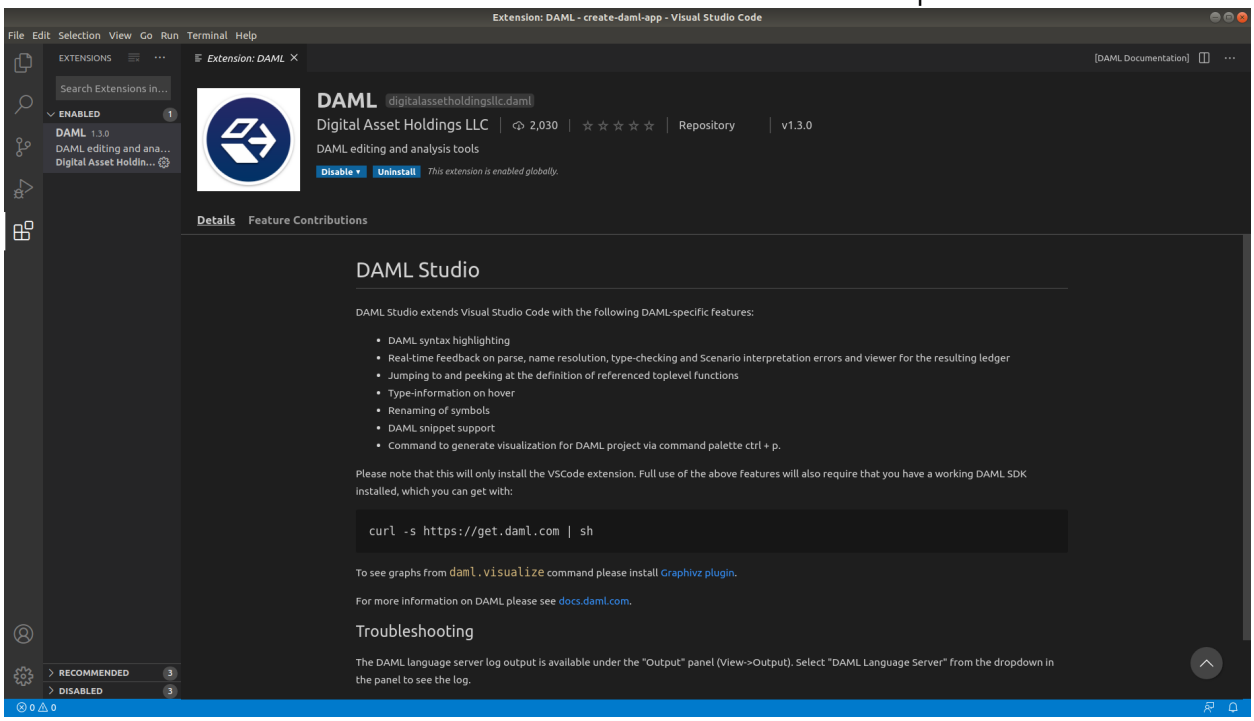
Daml Studio is an integrated development environment (IDE) for Daml. It is an extension on top of Visual Studio Code (VS Code), a cross-platform, open-source editor providing a rich code editing experience.

6.2.1 Installing

Daml Studio is included in *the Daml Connect SDK*.

6.2.2 Creating your first Daml file

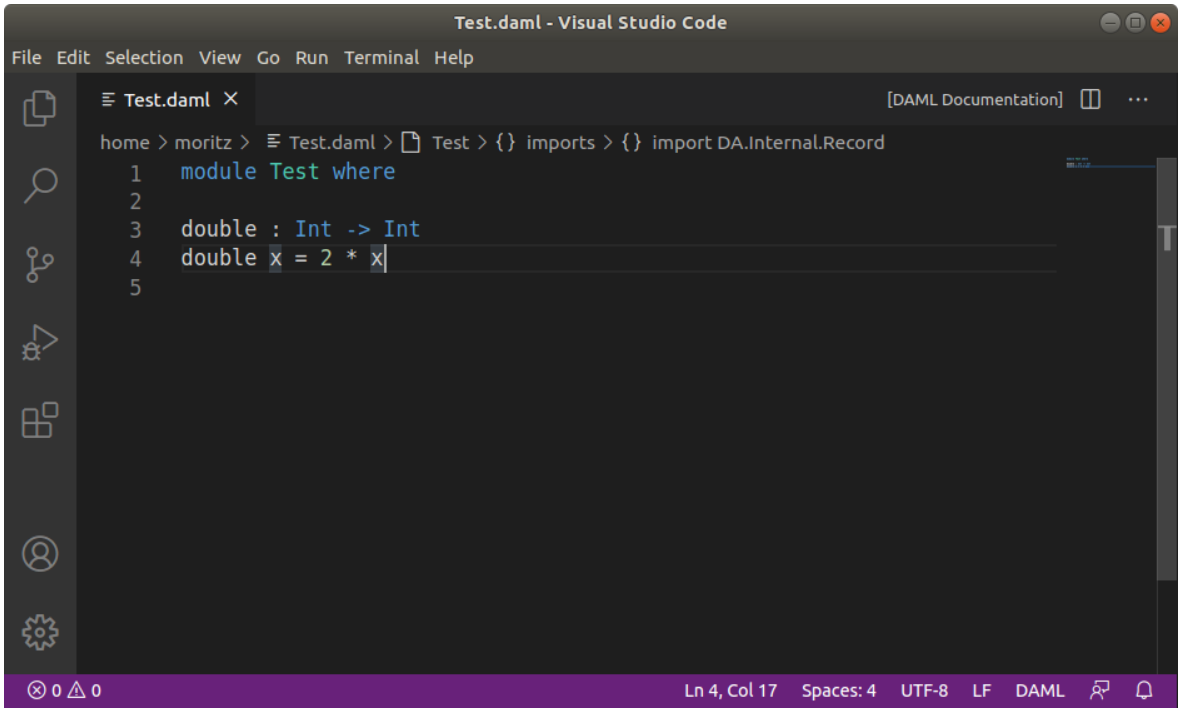
1. Start Daml Studio by running `daml studio` in the current project.
   This command starts Visual Studio Code and (if needs be) installs the Daml Studio extension, or upgrades it to the latest version.
2. Make sure the Daml Studio extension is installed:
   1. Click on the Extensions icon at the bottom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



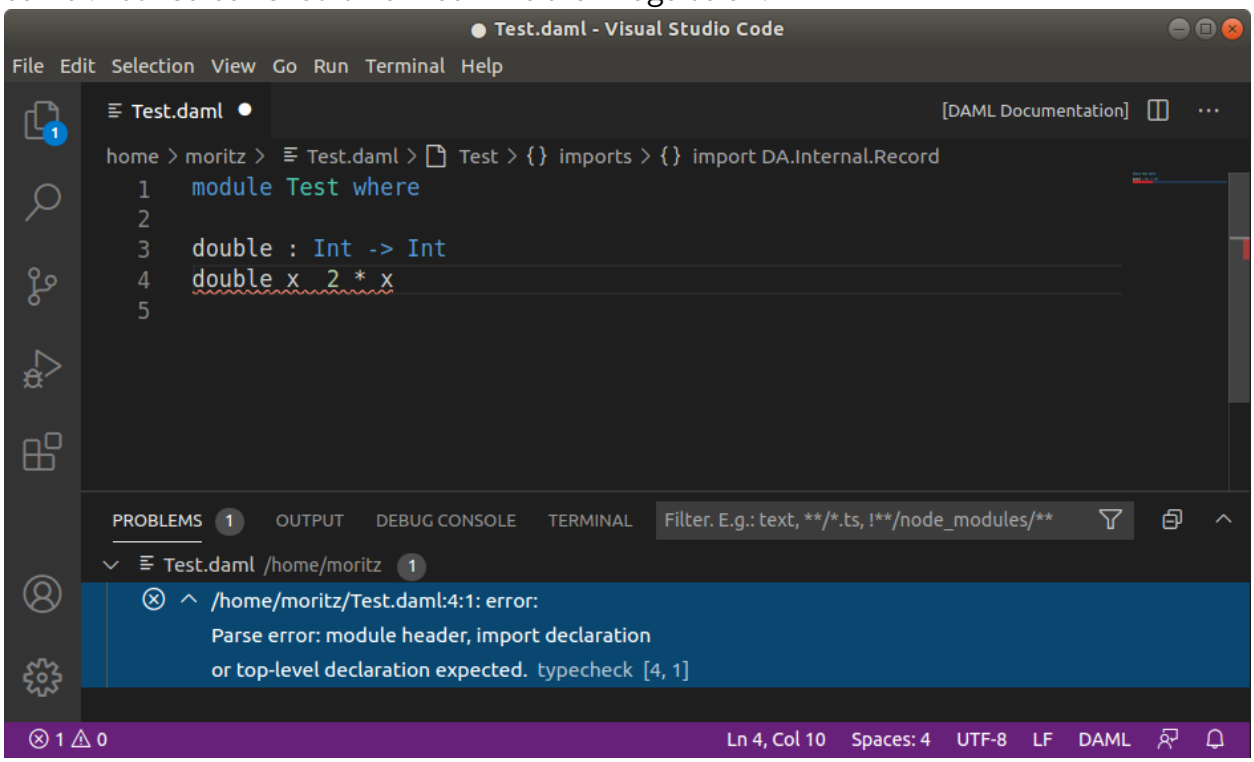
3. Open a new file (⌘N) and save it (⌘S) as `Test.daml`.
4. Copy the following code into your file:

```
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*.

6.2.3 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:

6.2.3.1 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

6.2.3.2 Hover tooltips

You can hover over most symbols in the code to display additional information such as its type.

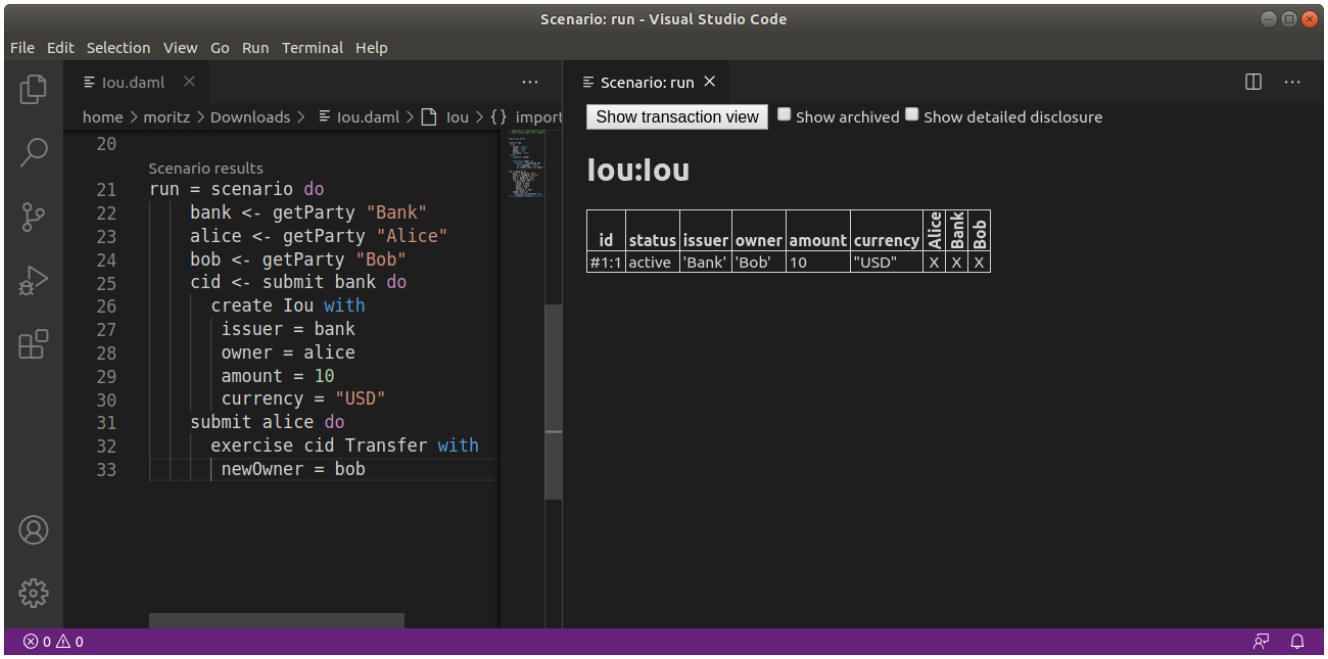
6.2.3.3 Scenario and Daml Script results

Top-level declarations of type `Scenario` or `Script` are decorated with a `Scenario results` or a `Script results` code lens. You can click on the code lens to inspect the execution transaction graph and the active contracts. The functionality for inspecting the results is identical for Daml Scripts and scenarios.

For the scenario from the `Iou` module, you get the following table displaying all contracts that are active at the end of the scenario. 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.

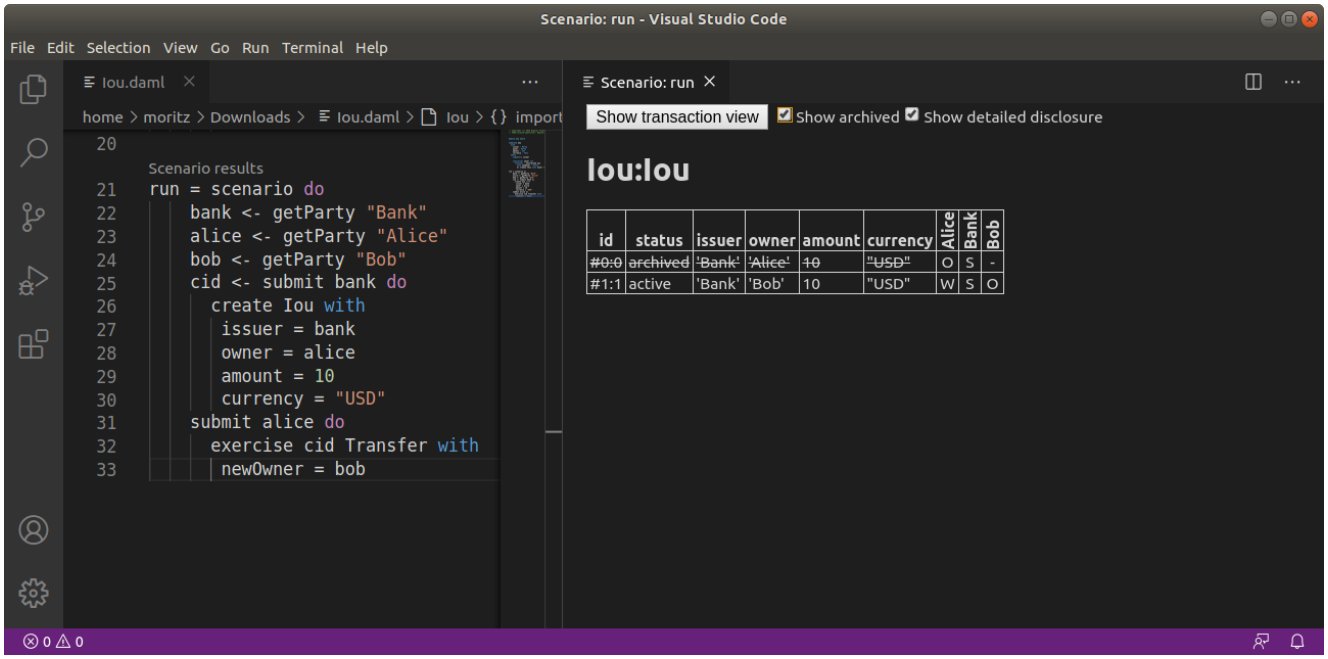
If you want more details, you can click on the *Show archived* checkbox, which extends the table to include archived contracts, and on the *Show detailed disclosure* checkbox, which displays why the contract is visible to each party, based on four categories:

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 scenario for the `Iou` module looks as follows:

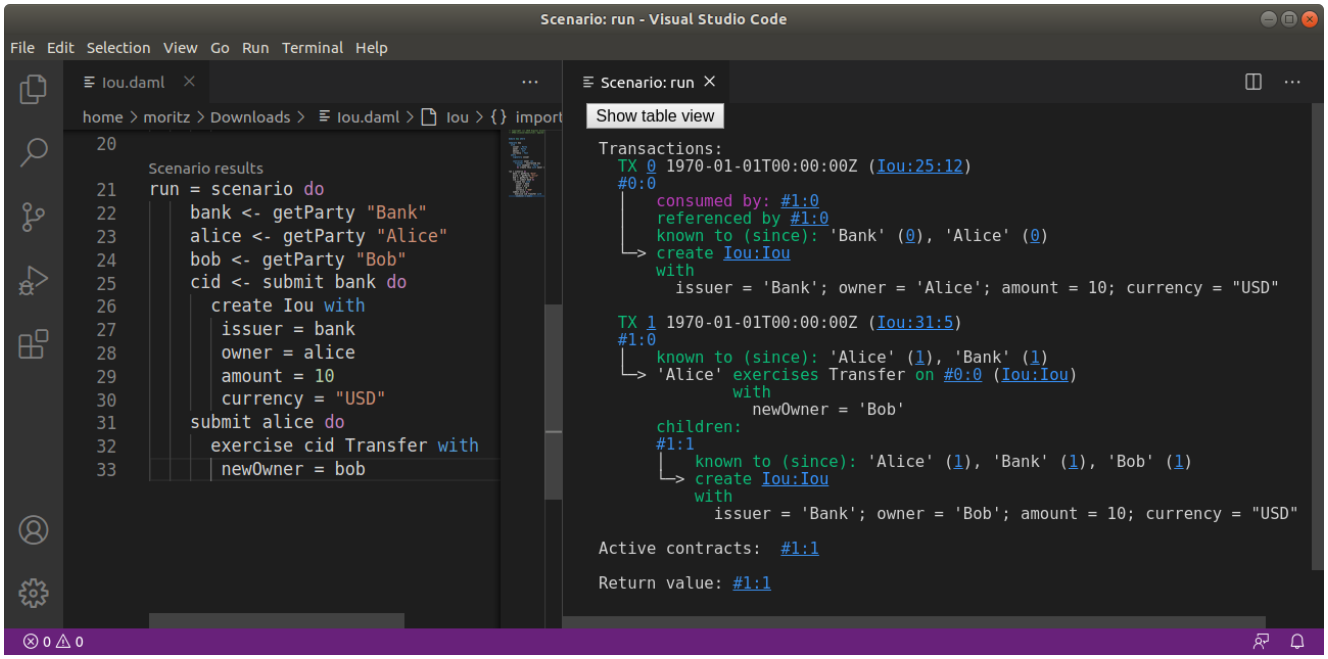


Fig. 1: Scenario results

Each transaction is the result of executing a step in the scenario. In the image below, the transaction `#0` is the result of executing the first line of the scenario (line 20), where the Iou is created by the bank. The following information can be gathered from the transaction:

> The result of the first scenario 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 scenario only the contract created in `#1:1` remains.
> The return value from running the scenario is the contract identifier `#1:1`.
> And finally, the contract identifiers assigned in scenario execution correspond to the scenario 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`).

6.2.3.4 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:

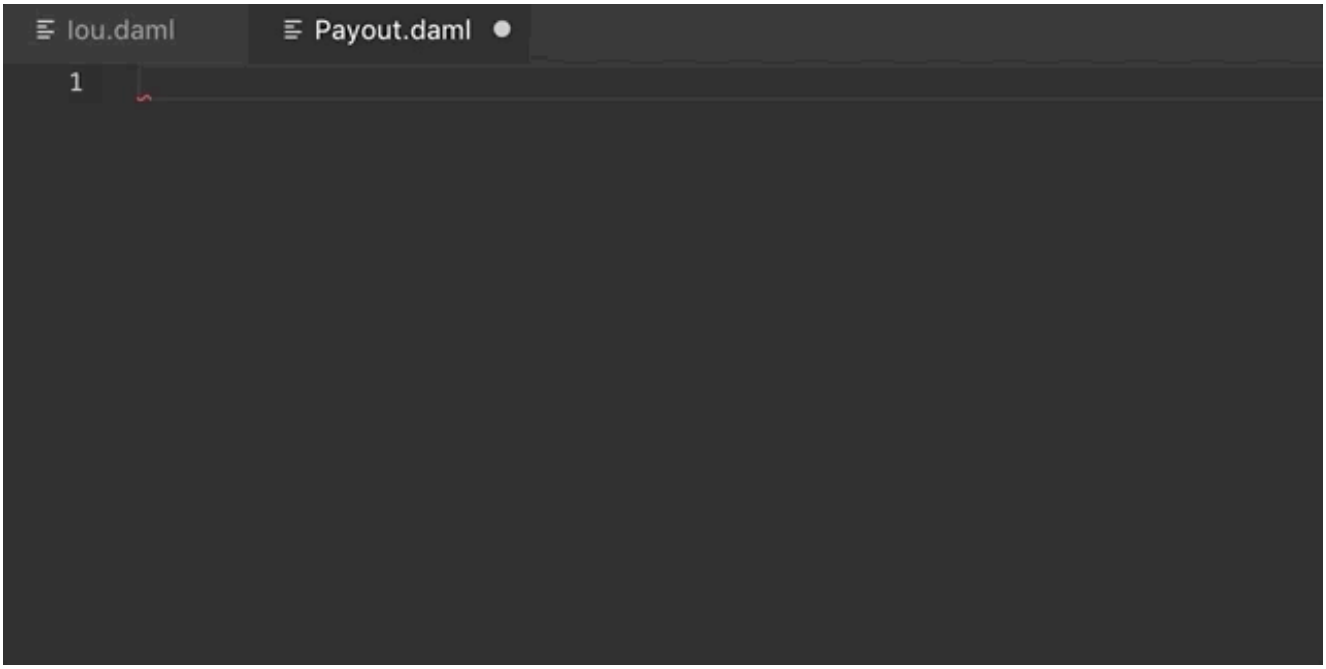
```
{
  "editor.tabCompletion": true,
```

(continues on next page)

(continued from previous page)

```
  "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.

6.2.4 Common scenario 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 scenario execution, the scenario 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 `submit`, the source location of the call to `submit` 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`.

6.2.4.1 Abort, assert, and debug

The `abort`, `assert` and `debug` inbuilt functions can be used in updates and scenarios. All three can be used to output messages, but `abort` and `assert` can additionally halt the execution:

```
abortTest = scenario do
  debug "hello, world!"
  abort "stop"
```

```
Scenario execution failed:
  Aborted:  stop

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

Partial transaction:

Trace:
  "hello, world!"
```

6.2.4.2 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 = scenario do
  alice <- getParty "Alice"
  bob <- getParty "Bob"
  submit alice (create Example with party1=alice; party2=bob)
```

Execution of the example scenario 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.

6.2.4.3 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

    controller owner can
      Consume : ()
        do return ()

    controller friend can
      Hello : ()
        do return ()

example = scenario do
  alice <- getParty "Alice"
  bob <- getParty "Bob"
  cid <- submit alice (create Example with owner=alice; friend=bob)
  submit bob do exercise cid Consume
```

The execution of the example scenario 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:
  Sub-transactions:
     #0
     └─> fetch #0:0 (ExerciseAuthFailure:Example)

     #1
     └─> '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.

6.2.4.4 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

    controller owner can
      Consume : ()
        do return ()

example = scenario do
  alice <- getParty "Alice"
  bob <- getParty "Bob"
  cid <- submit alice (create Example with owner=alice)
  submit bob do exercise cid Consume
```

In the above scenario 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 committer.
  Contract:  #0:0 (NotVisibleFailure:Example)
  Committer: 'Bob'
  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.

6.2.5 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.

6.3 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`.

Note: Sandbox has switched to use Wall Clock Time mode by default. To use Static Time Mode you can provide the `--static-time` flag to the `daml sandbox` command or configure the time mode for `daml start` in `sandbox-options:` section of `daml.yaml`. Please refer to *Daml configuration files* for more information.

Sandbox can also be run manually as in this example:

```
$ daml sandbox Main.dar --static-time


   ____              ____
  / __/__ ____  ___/ / /  ___ __ __
 _\ \/ _ `/ _ \/ _  / _ \/ _ \\ \ /
/___/\_,_/_//_/\_,_/_.__/\___/_\_\
```

(continues on next page)

(continued from previous page)

```
INFO: Initialized sandbox version 1.12.0-snapshot.20210312.6498.0.707c86aa
↪with ledger-id = fd562651-5ebb-4a45-add7-25809ca1f297, port = 6865, dar
↪file = List(Main.dar), time mode = static time, ledger = in-memory, auth-
↪service = AuthServiceWildcard$, contract ids seeding = strong
```

6.3.1 Contract Identifier Generation

Sandbox supports two contract identifier generator schemes:

> The so-called *deterministic* scheme that deterministically produces contract identifiers from the state of the underlying ledger. Those identifiers are strings starting with `#`.
> The so-called *random* scheme that produces contract identifiers indistinguishable from random. In practice, the schemes use a cryptographically secure pseudorandom number generator initialized with a truly random seed. Those identifiers are hexadecimal strings prefixed by `00`.

The sandbox can be configured to use one or the other scheme with one of the following command line options:

> `--contract-id-seeding=<seeding-mode>`. The Sandbox will use the seeding mode *<seeding-mode>* to seed the generation of random contract identifiers. Possible seeding modes are:
> - `no`: The Sandbox uses the `deterministic` scheme. This is only supported by Sandbox classic and it prevents Sandbox from accepting packages in Daml-LF 1.11 or newer.
> - `strong`: The Sandbox uses the `random` scheme initialized with a high-entropy seed. Depending on the underlying operating system, the startup of the Sandbox may block as entropy is being gathered to generate the seed.
> - `testing-weak`: (**For testing purposes only**) The Sandbox uses the `random` scheme initialized with a low entropy seed. This may be used in a testing environment to avoid exhausting the system entropy pool when a large number of Sandboxes are started in a short time interval.
> - `testing-static`: (**For testing purposes only**) The sandbox uses the `random` scheme with a fixed seed. This may be used in testing for reproducible runs.

6.3.2 Running with persistence

Note: Running Sandbox with persistence is deprecated as of SDK 1.8.0 (16th Dec 2020). You can use the Daml Driver for PostgreSQL instead.

By default, Sandbox uses an in-memory store, which means it loses its state when stopped or restarted. If you want to keep the state, you can use a Postgres database for persistence. This allows you to shut down Sandbox and start it up later, continuing where it left off.

To set this up, you must:

> create an initially empty Postgres database that the Sandbox application can access
> have a database user for Sandbox that has authority to execute DDL operations
> This is because Sandbox manages its own database schema, applying migrations if necessary when upgrading versions.

To start Sandbox using persistence, pass an `--sql-backend-jdbcurl <value>` option, where `<value>` is a valid jdbc url containing the username, password and database name to connect to.

Here is an example for such a url: `jdbc:postgresql://localhost/test?user=fred&password=secret`

Due to possible conflicts between the `&` character and various terminal shells, we recommend quoting the jdbc url like so:

```
$ daml sandbox Main.dar --sql-backend-jdbcurl "jdbc:postgresql://localhost/
↪test?user=fred&password=secret"
```

If you're not familiar with JDBC URLs, see the JDBC docs for more information: https://jdbc.postgresql.org/documentation/head/connect.html

6.3.3 Running with authentication

By default, Sandbox does not use any authentication and accepts all valid ledger API requests.

To start Sandbox with authentication based on JWT tokens, use one of the following command line options:

`--auth-jwt-rs256-crt=<filename>`. 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.
`--auth-jwt-es256-crt=<filename>`. 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.
`--auth-jwt-es512-crt=<filename>`. 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.
`--auth-jwt-rs256-jwks=<url>`. 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.

Warning: For testing purposes only, the following options may also be used. None of them is considered safe for production:

`--auth-jwt-hs256-unsafe=<secret>`. The sandbox will expect all tokens to be signed with HMAC256 with the given plaintext secret.

6.3.3.1 Token payload

JWTs express claims which are documented in the *authorization* documentation.

The following is an example of a valid JWT payload:

```
{
   "https://daml.com/ledger-api": {
     "ledgerId": "aaaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee",
     "participantId": null,
     "applicationId": null,
     "admin": true,
     "actAs": ["Alice"],
```

(continues on next page)

(continued from previous page)

```
      "readAs": ["Bob"]
    },
    "exp": 1300819380
}
```

where

> `ledgerId`, `participantId`, `applicationId` restricts the validity of the token to the given ledger, participant, or application
> `exp` is the standard JWT expiration date (in seconds since EPOCH)
> `admin`, `actAs` and `readAs` bear the same meaning as in the *authorization* documentation

The `public` claim is implicitly held by anyone bearing a valid JWT (even without being an admin or being able to act or read on behalf of any party).

6.3.3.2 Generating JSON Web Tokens (JWT)

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

6.3.3.3 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

6.3.3.4 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
```

6.3.4 Running with TLS

To enable TLS, you need to specify the private key for your server and the certificate chain via `daml sandbox --pem server.pem --crt server.crt`. By default, Sandbox requires client authentication as well. You can set a custom root CA certificate used to validate client certificates via `--`

`cacrt ca.crt`. You can change the client authentication mode via `--client-auth none` which will disable it completely, `--client-auth optional` which makes it optional or specify the default explicitly via `--client-auth require`.

6.3.5 Command-line reference

To start Sandbox, run: `sandbox [options] <archive>...`.

To see all the available options, run `daml sandbox --help`.

6.3.6 Metrics

6.3.6.1 Enable and configure reporting

To enable metrics and configure reporting, you can use the two following CLI options:

`--metrics-reporter`: passing a legal value will enable reporting; the accepted values are `console`, `csv:</path/to/metrics.csv>` and `graphite:<local_server_port>`.

- `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.

`--metrics-reporting-interval`: metrics are pre-aggregated on the sandbox 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.

6.3.6.2 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

6.3.6.3 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.

6.4 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.

6.4.1 Navigator functionality

Connect 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.)

6.4.2 Installing and starting Navigator

Navigator ships with 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

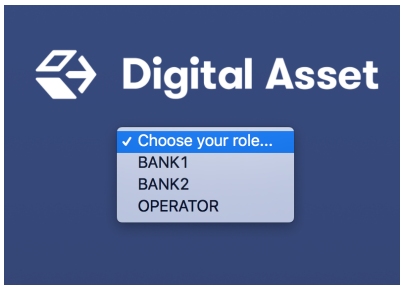
   When running `daml start` you will see the Navigator URL. By default it will be http://localhost:7500/.

Note: Navigator is compatible with these browsers: Safari, Chrome, or Firefox.

For information on how to launch and use Navigator outside of the SDK, see *Advanced usage* below.

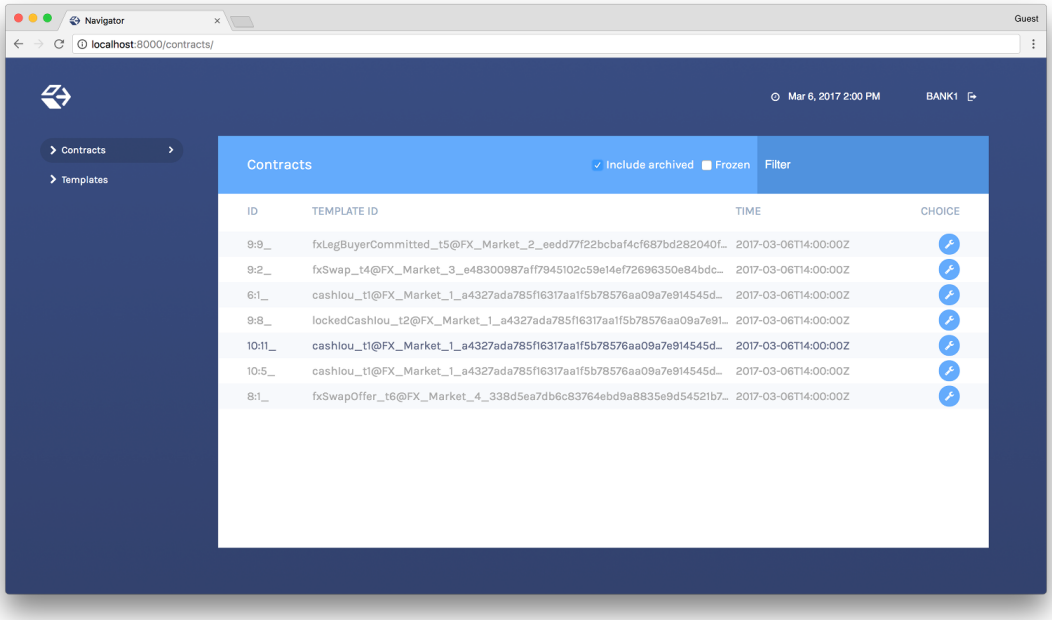
6.4.3 Choosing a party / changing the party

The ledger is a record of transactions between authorized participants on the distributed network. Before you can interact with the ledger, you must assume the role of a particular party. This determines the contracts that you can access and the actions you are permitted to perform on the ledger. The first step in using Navigator is to use the drop-down list on the Navigator home screen to select from the available parties.



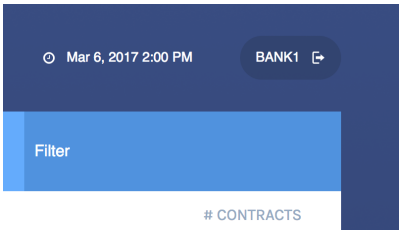
Note: The party choices are configured on startup. (Refer to *Daml Assistant (daml)* or *Advanced usage* for more instructions.)

The main Navigator screen will be displayed, with contracts that this party 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 party:

1. Click the name of the current party in the top right corner of the screen.
2. On the home screen, select a different party.



You can act as different parties in different browser windows. Use Chrome's profile feature https://support.google.com/chrome/answer/2364824 and sign in as a different party for each Chrome profile.

6.4.4 Logging out

To log out, click the name of the current party in the top-right corner of the screen.

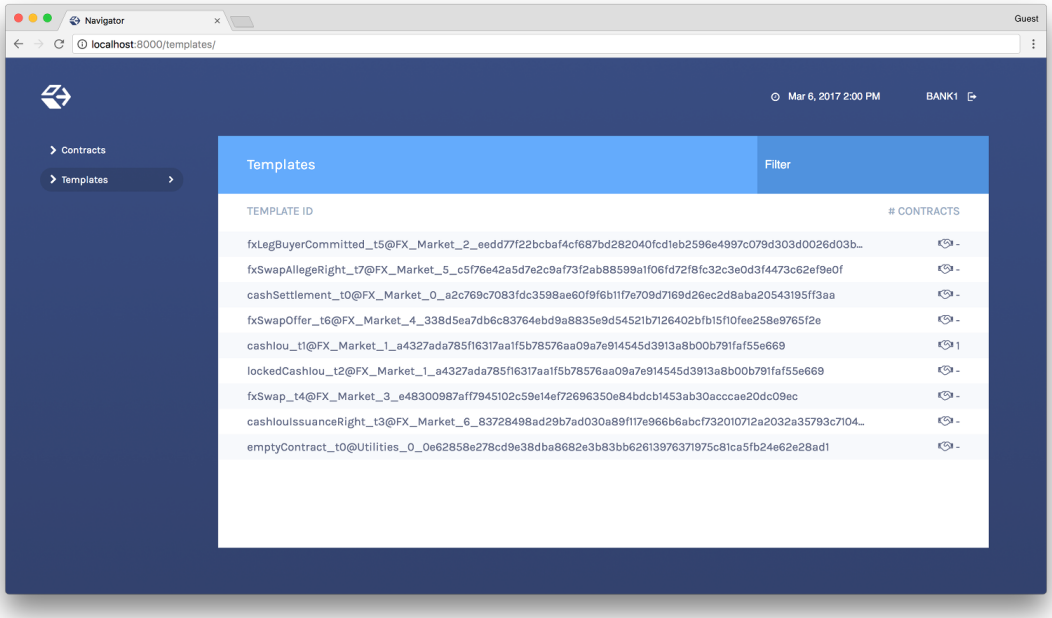
6.4.5 Viewing templates or contracts

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.

6.4.5.1 Listing templates

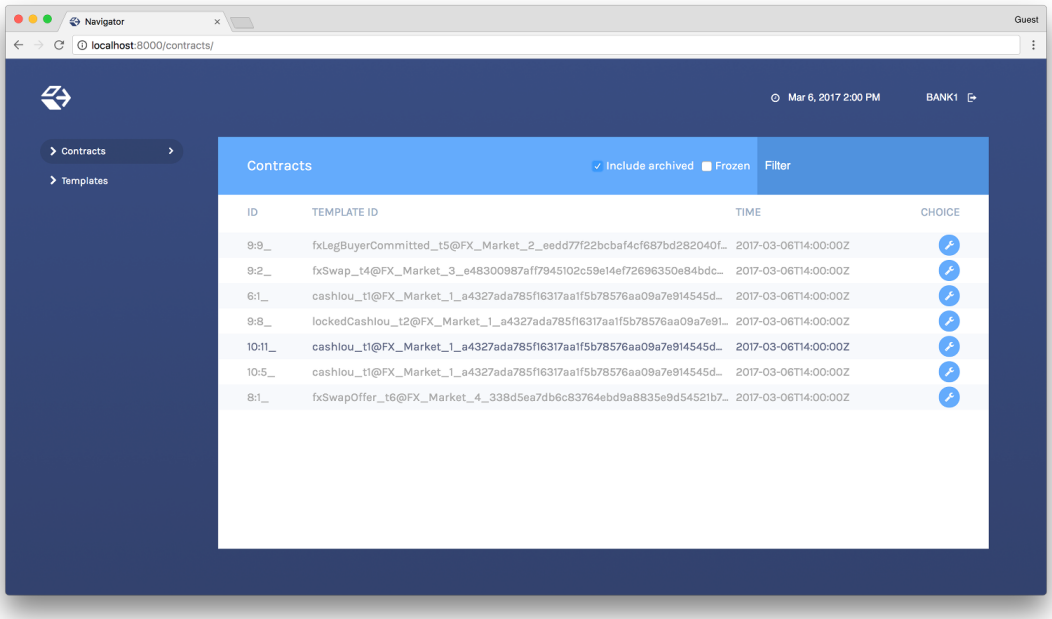
To see what contract templates are available on the ledger you are connected to, choose **Templates** in the left pane of the main Navigator screen.



Use the **Filter** field at the top right to select template IDs that include the text you enter.

6.4.5.2 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.

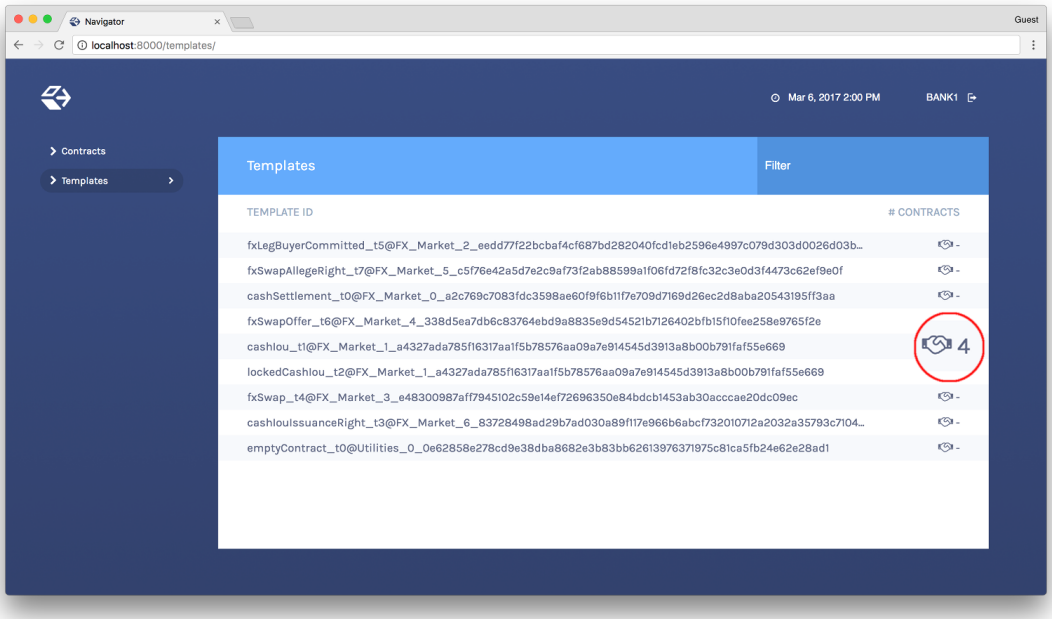
Filter the displayed contracts by entering text in the **Filter** field at the top right.
Use the **Include Archived** checkbox at the top to include or exclude archived contracts.

6.4.5.3 Viewing contracts based on a template

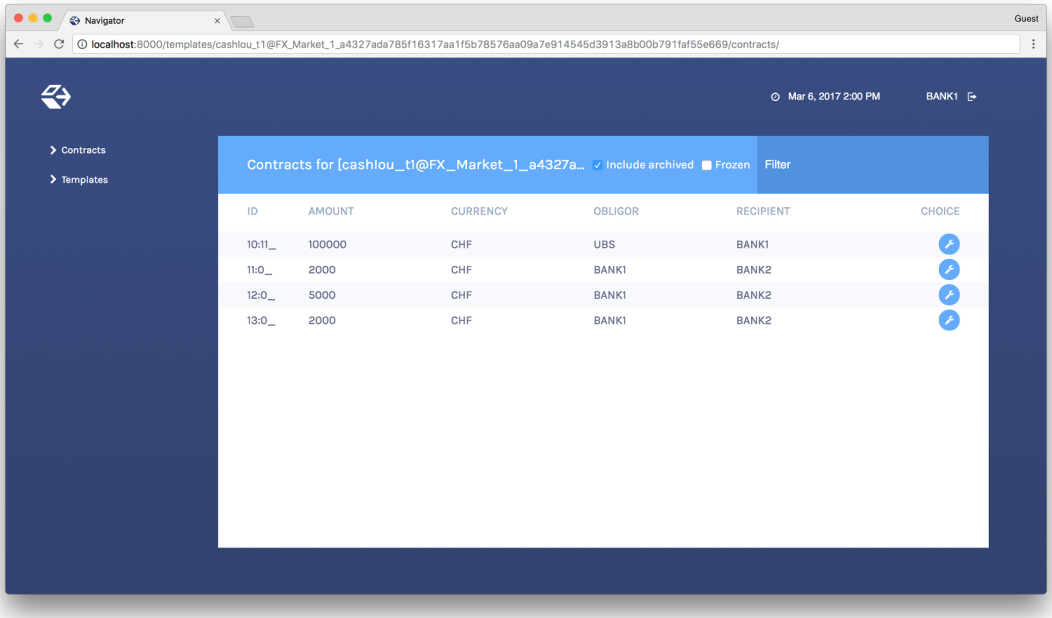
You can also view the list of contracts that are based on a particular template.

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



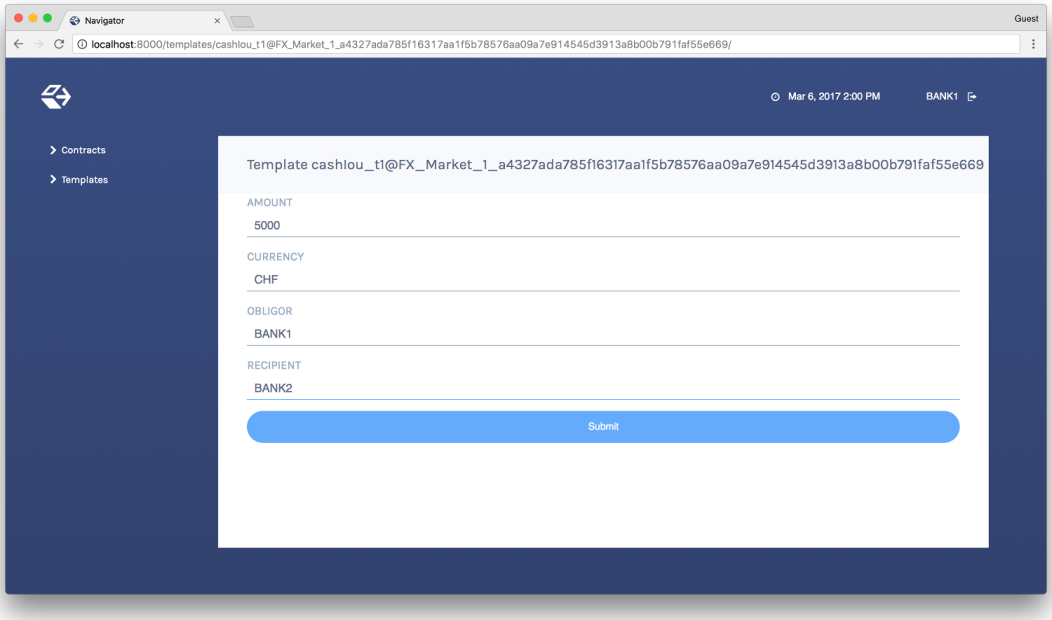
List of Contracts



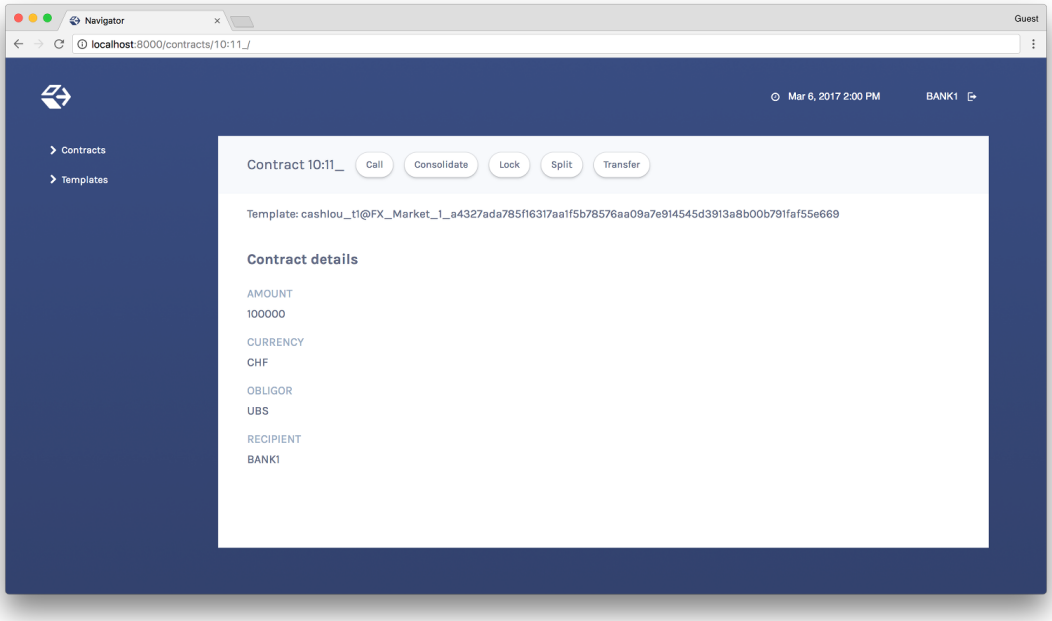
6.4.5.4 Viewing template and contract details

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



6.4.6 Using Navigator

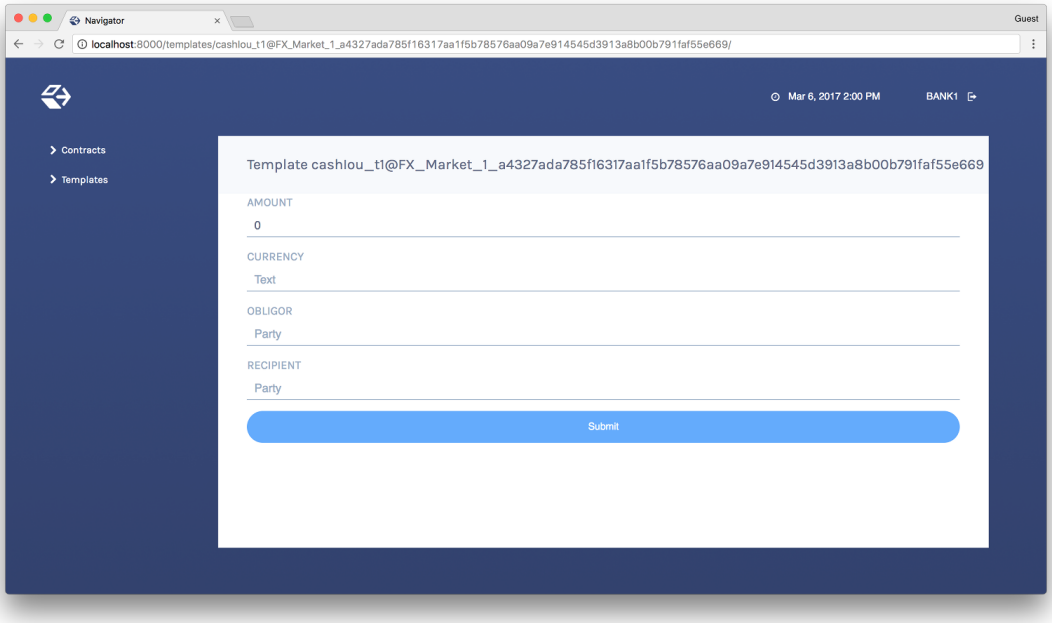
6.4.6.1 Creating contracts

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 experi-

menting during development.

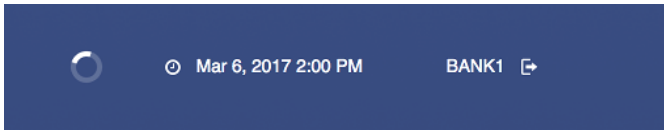
To create a contract based on a template:

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

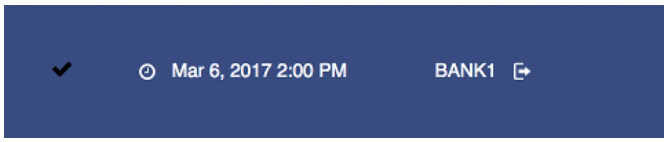


When the command has been committed to the ledger, the loading indicator in the navbar at the top will display a tick mark.

While loading



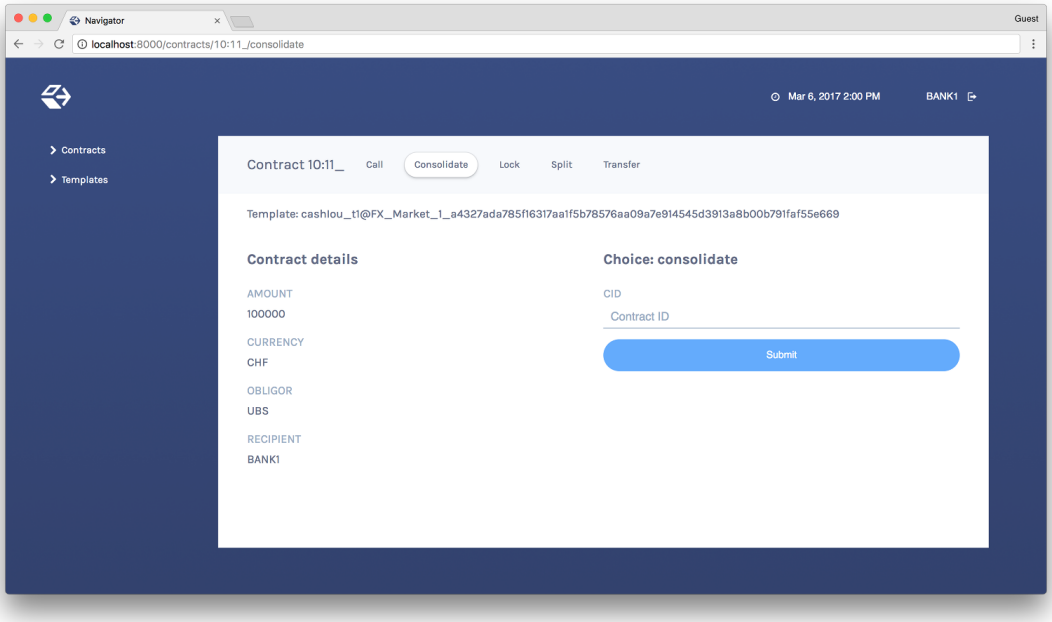
When committed to the ledger



6.4.6.2 Exercising choices

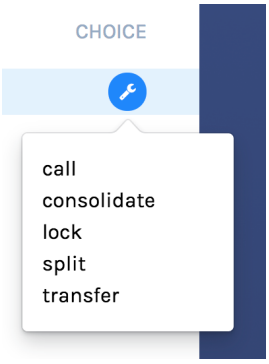
To exercise a choice:

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.



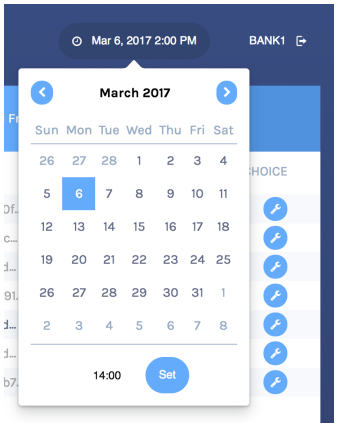
You will see the loading and confirmation indicators, as pictured above in Creating Contracts.

6.4.6.3 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.



6.4.7 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:

6.4.8 Advanced usage

6.4.8.1 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,
};
```

(continues on next page)

(continued from previous page)

```
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}) => ({
          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).

6.4.8.2 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
```

6.5 Daml codegen

6.5.1 Introduction

You can use the Daml codegen to generate Java, Scala, 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`.

6.5.2 Running the Daml codegen

The basic command to run the Daml codegen is:

```
$ daml codegen [java|scala|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** and **Scala** only)

6.5.2.1 Command line configuration

Help for each specific codegen:

```
$ daml codegen [java|scala|js] --help
```

Java and **Scala** codegens take the same set of configuration settings:

```
<DAR-file[=package-prefix]>...
                          DAR file to use as input of the codegen with an
↪optional, but recommend, package prefix for the generated sources.
-o, --output-directory <value>
                          Output directory for the generated sources
-d, --decoderClass <value>
                          Fully Qualified Class Name of the optional
↪Decoder utility
```

(continues on next page)

(continued from previous page)

```
-V, --verbosity <value>  Verbosity between 0 (only show errors) and 4
↪(show all messages) -- defaults to 0
-r, --root <value>       Regular expression for fully-qualified names of
↪templates to generate -- defaults to .*
--help                   This help text
```

JavaScript/TypeScript codegen takes a different set of configuration settings:

```
DAR-FILES                DAR files to generate TypeScript bindings for
-o DIR                   Output directory for the generated packages
-s SCOPE                 The NPM scope name for the generated packages;
                        defaults to daml.js
-h,--help                Show this help text
```

6.5.2.2 Project file configuration (Java and Scala)

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: 1.2.0
name: quickstart
source: daml
scenario: Main:setup
parties:
  - Alice
  - Bob
  - USD_Bank
  - EUR_Bank
version: 0.0.1
exposed-modules:
  - Main
dependencies:
  - daml-prim
  - daml-stdlib
codegen:
  js:
    output-directory: ui/daml.js
    npm-scope: daml.js
  java:
    package-prefix: com.daml.quickstart.iou
    output-directory: java-codegen/src/main/java
    verbosity: 2
  scala:
    package-prefix: com.daml.quickstart.iou
    output-directory: scala-codegen/src/main/scala
    verbosity: 2
```

You can then run the above configuration to generate your **Java** or **Scala** code:

```
$ daml codegen [js|java|scala]
```

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** or **Scala** command line configuration:

```
$ daml codegen [java|scala| ./.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** or **Scala** classes, you need to add the corresponding dependencies to your build tools.

For **Scala**, you can depend on:

```
"com.daml" %% "bindings-scala" % YOUR_SDK_VERSION
```

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

6.6 Daml Profiler

The Daml Profiler is only available in the Daml Connect Enterprise Edition.

The Daml Profiler allows you to to profile execution of your Daml code which can help spot bottlenecks and opportunities for optimization.

6.6.1 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.

```
daml sandbox .daml/dist/profile-tutorial-0.0.1.dar --profile-dir profile-
↪results
```

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
```

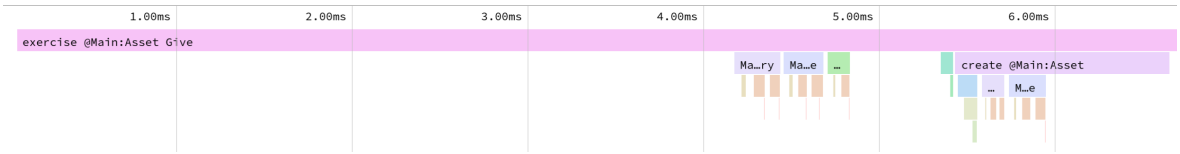
If we now look at the contents of the `profile-results` directory, we can see one JSON file per transaction produced by the script. Each file has a name of the form `$timestamp-$command.json` where `$timestamp` is the submission time of the transaction and `$command` is a human-readable description of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

6.6.2 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 7

Background concepts

7.1 Glossary of concepts

7.1.1 Daml

Daml is a programming language for writing *smart contracts*, that you can use to build an application based on a *ledger*. You can run Daml contracts on many different ledgers.

7.1.1.1 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.

7.1.1.2 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*.

7.1.1.3 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.

7.1.1.4 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 in Daml.

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.

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*.

7.1.1.5 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.

7.1.1.6 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*.

7.1.1.7 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*.

7.1.1.8 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*.

7.1.1.9 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*.

7.1.1.10 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*.

7.1.1.11 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*.

7.1.1.12 DAR file, DALF file

A `.dar` file is the result of compiling Daml using the *Assistant*.

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*.

7.1.2 Developer tools

7.1.2.1 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)*.

7.1.2.2 Studio

Daml Studio is a plugin for Visual Studio Code, and is the IDE for writing Daml code.

See *Daml Studio*.

7.1.2.3 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*.

7.1.2.4 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*.

7.1.2.5 Extractor

Extractor is a tool for extracting contract data for a single party into a PostgreSQL database.

See *Extractor*.

7.1.3 Building applications

7.1.3.1 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.

7.1.3.2 Ledger API

The **Ledger API** is an API that's exposed by any *Daml ledger*. Alternative names: **Daml Ledger API** and **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*.

Ledger configuration service

Use the **ledger configuration service** to subscribe to changes in *ledger* configuration. See *Ledger configuration service*.

7.1.3.3 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*.

Scala bindings

An idiomatic Scala library for writing *ledger applications*. See *Scala bindings*.

7.1.3.4 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.

7.1.3.5 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 *scenarios*, 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*.

7.1.3.6 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)

7.1.4 General concepts

7.1.4.1 Ledger, Daml ledger

Ledger can refer to a lot of things, but a ledger is essentially the underlying storage mechanism for a running Daml applications: it's where the contracts live. A **Daml ledger** is a ledger that you can store Daml contracts on, because it implements the *ledger API*.

Daml ledgers provide various guarantees about what you can expect from it, all laid out in the *Daml Ledger Model* page.

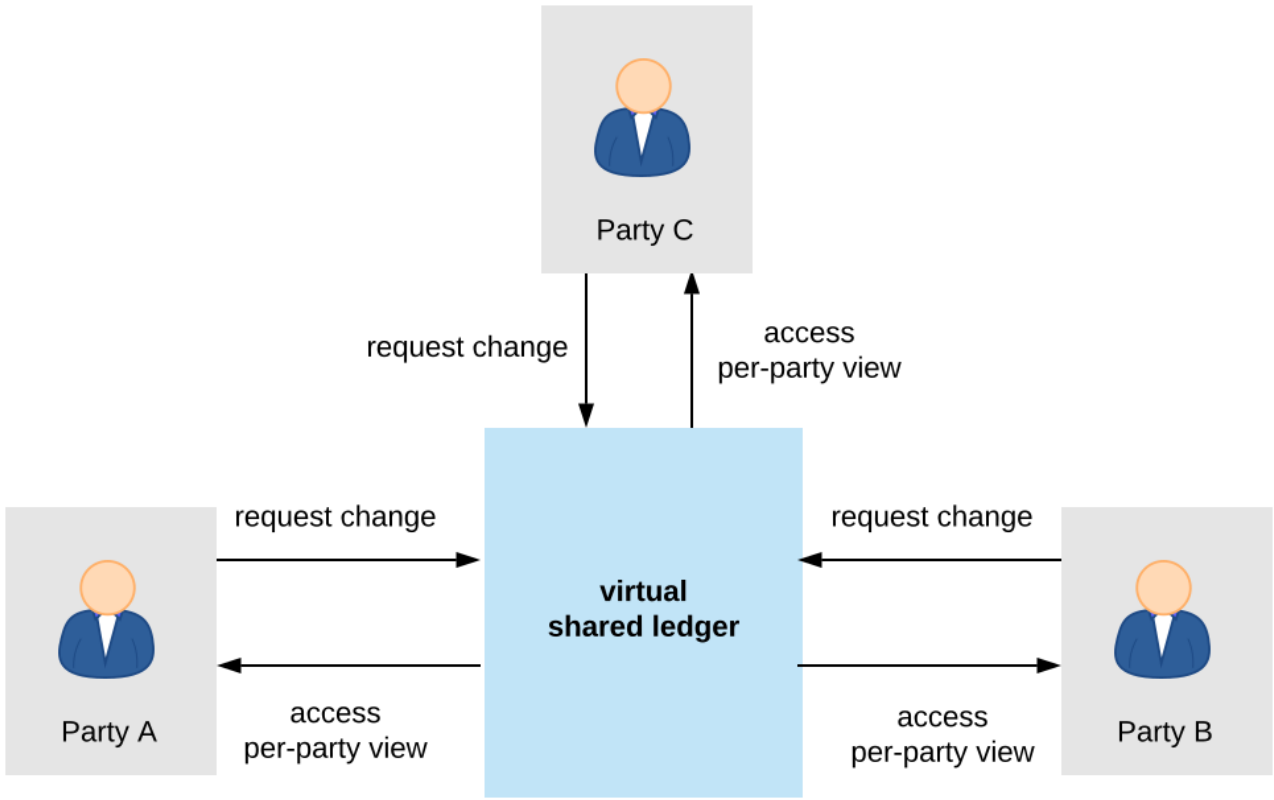
When you're developing, you'll use *Sandbox* as your ledger.

7.1.4.2 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.

7.2 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.

7.2.1 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*.

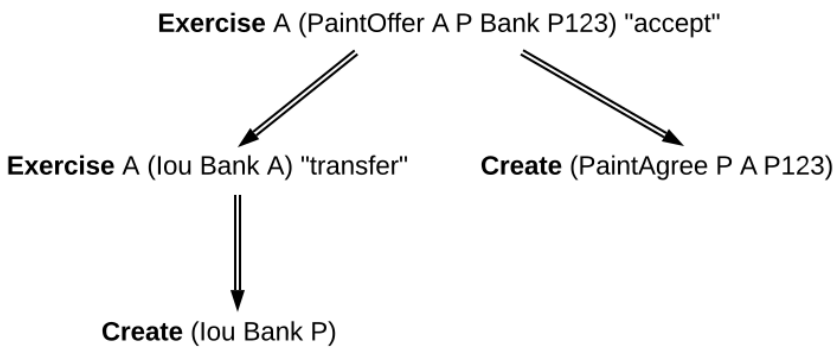
7.2.1.1 Actions and Transactions

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.

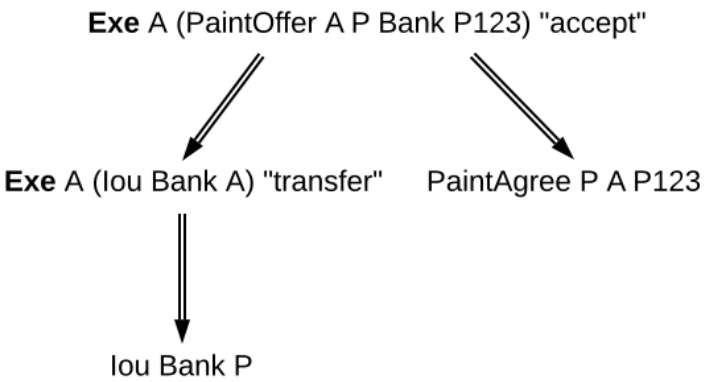
The following EBNF-like grammar summarizes the structure of actions and transactions. Here, s | t represents the choice between *s* and *t*, s t represents *s* followed by *t*, and s* represents the repetition of *s* zero or more times. The terminal 'contract' denotes the underlying type of contracts, 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'
```

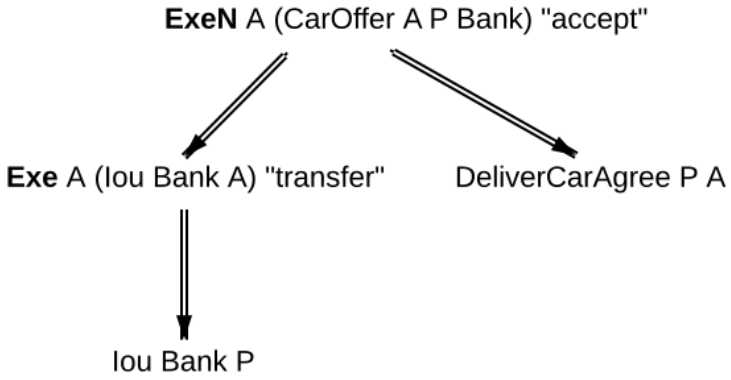
The visual notation presented earlier captures actions precisely with conventions that:

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.

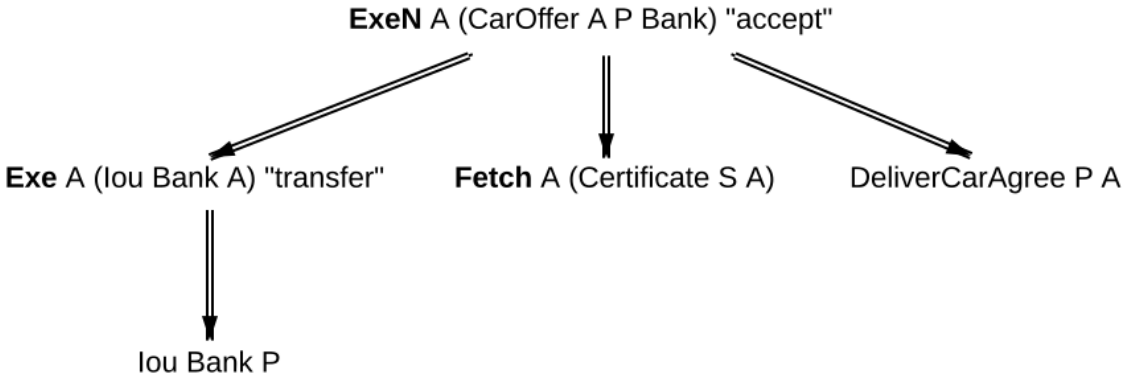
An alternative shorthand notation, shown below uses the abbreviations **Exe** and **ExeN** for exercises, and omits the **Create** labels on create actions.



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:

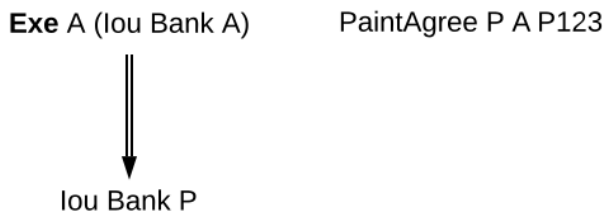
PaintOffer houseOwner painter obligor refNo Intuitively an offer (with a reference number) by which the painter proposes to the house owner to paint her house, in exchange for a single IOU token issued by the specified obligor.

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).

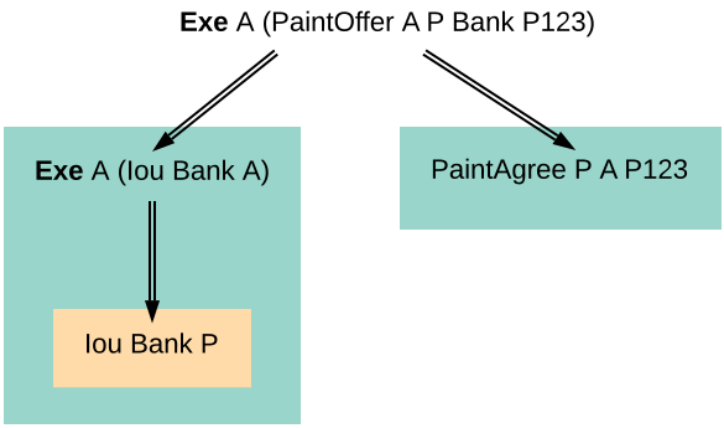
In practice, multiple IOU contracts can exist between the same *obligor* and *owner*, in which case each contract should have a unique identifier. However, in this section, each contract only appears once, allowing us to drop the notion of identifiers for simplicity reasons.

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.

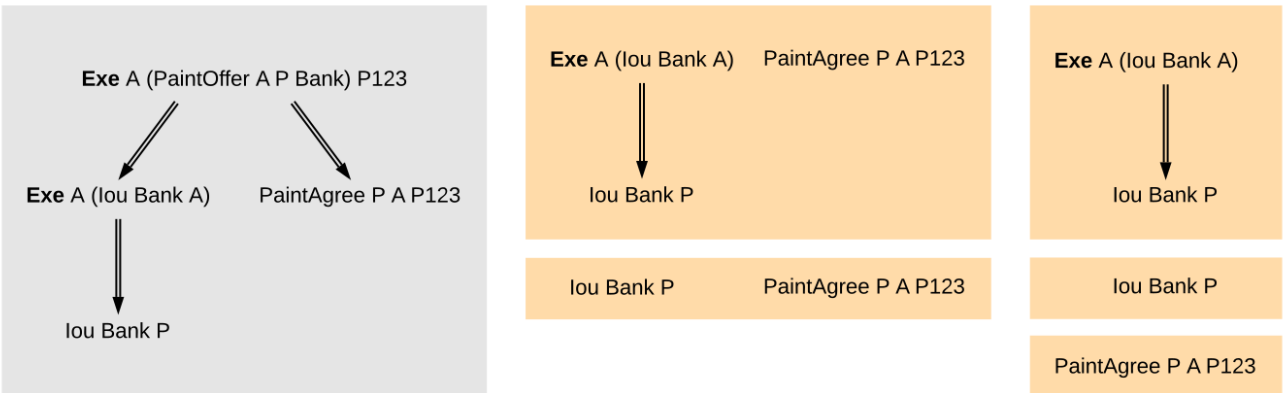


For an action *act*, its **proper subactions** are all actions in the consequences of *act*, together with all of their proper subactions. Additionally, *act* is a (non-proper) **subaction** of itself.

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.

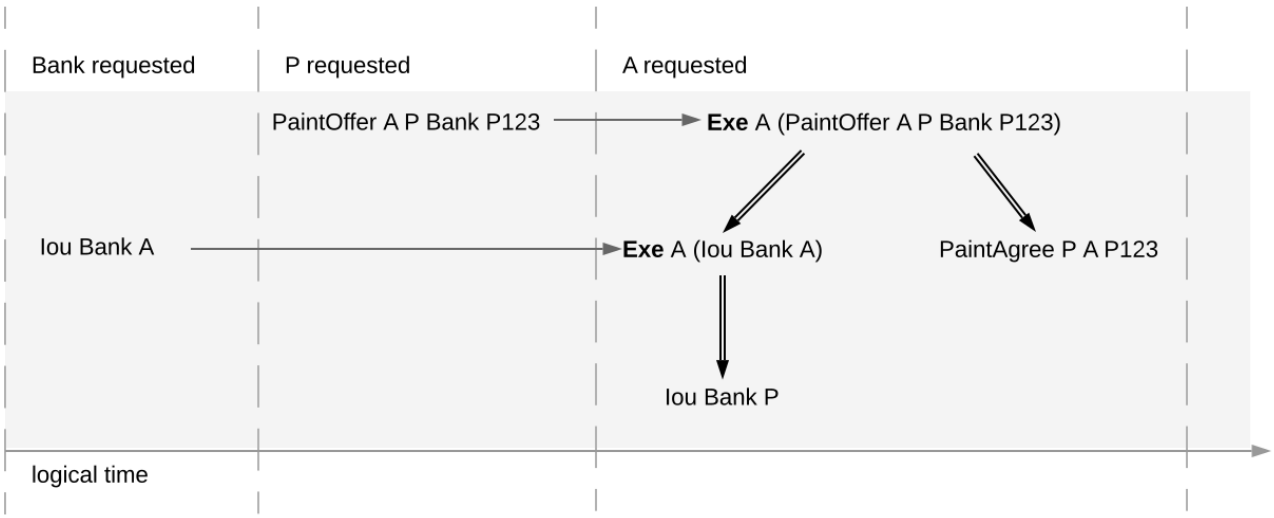
7.2.1.2 Ledgers

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.

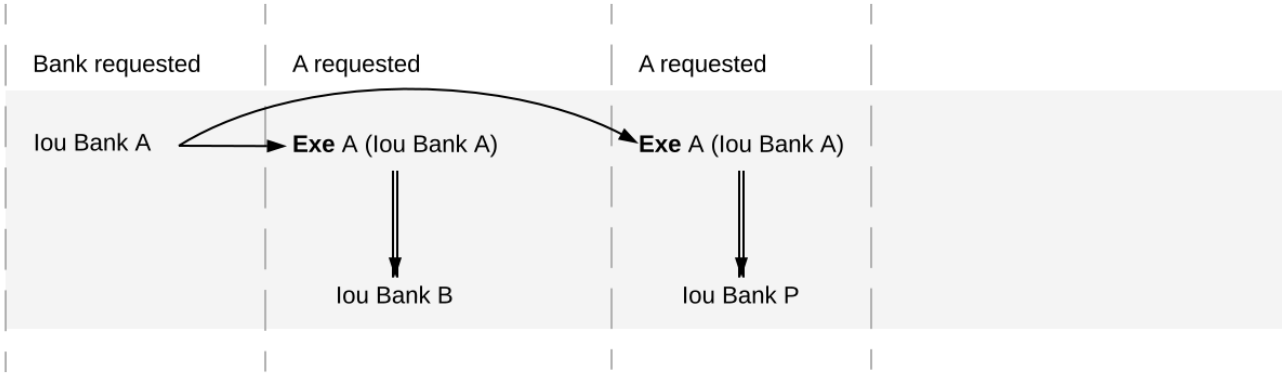


Fig. 1: Alice spending her IOU twice (double spend), once transferring it to *B* and once to *P*.

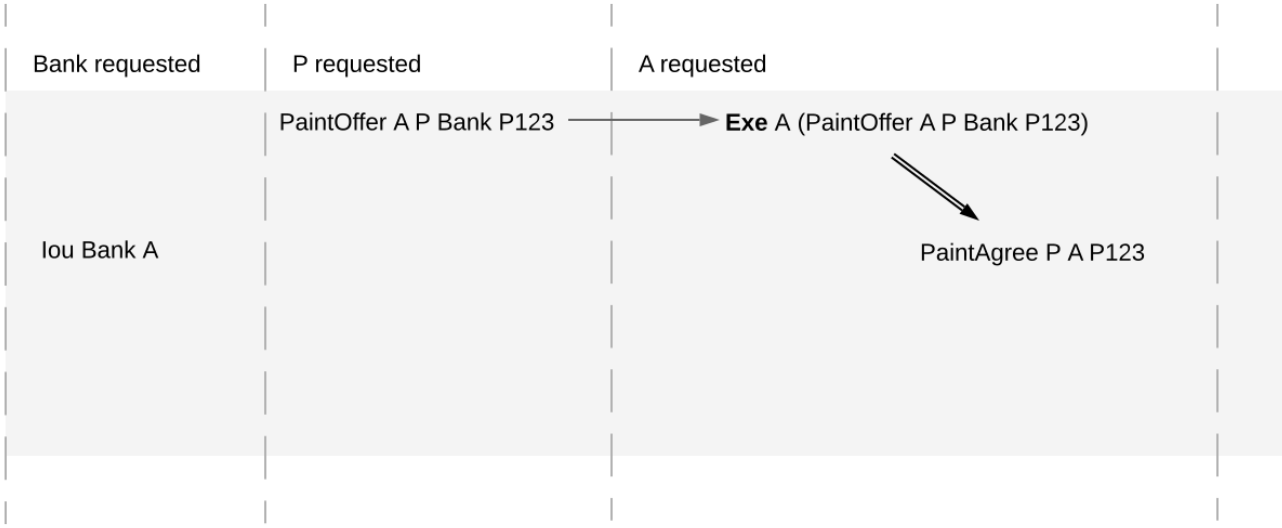


Fig. 2: Alice changing the offer's outcome by removing the transfer of the *Iou*.

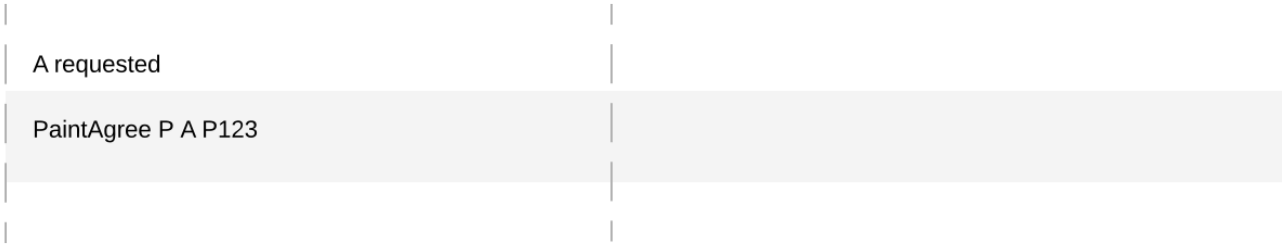


Fig. 3: An obligation imposed on the painter without his consent.

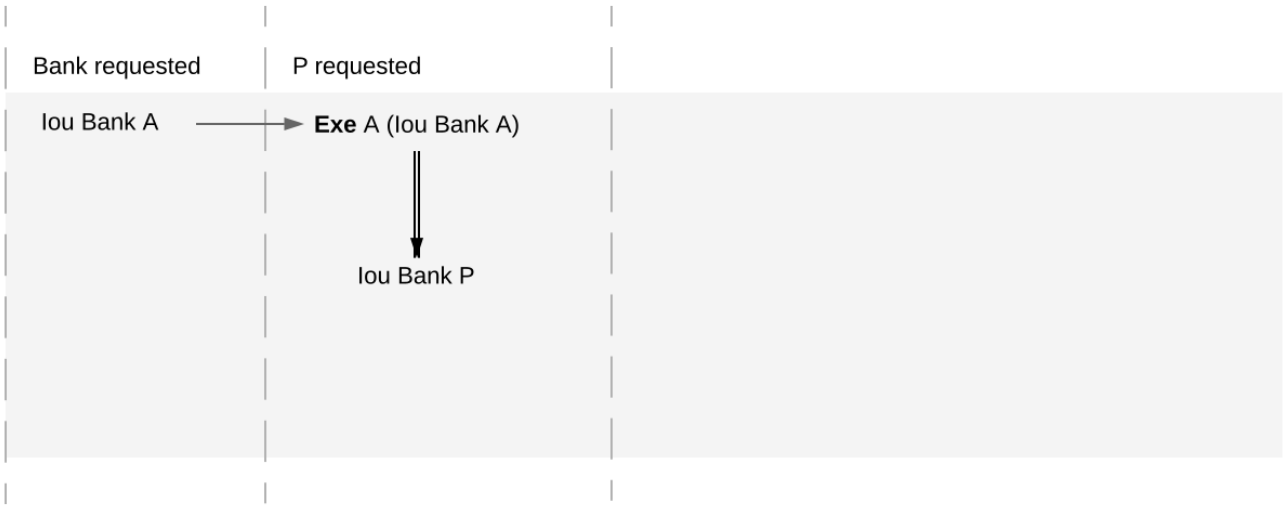


Fig. 4: Painter stealing Alice's IOU. Note that the ledger would be intuitively permissible if it was Alice performing the last commit.

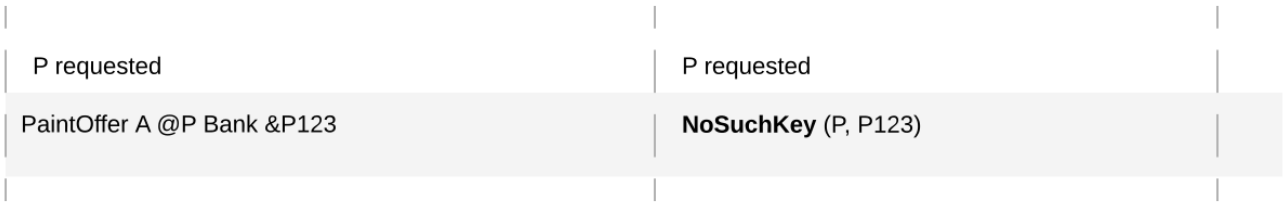


Fig. 5: Painter falsely claiming that there is no offer.

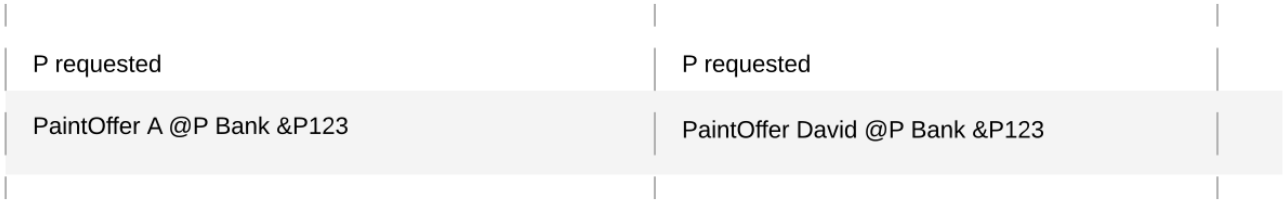


Fig. 6: Painter trying to create two different paint offers with the same reference number.

7.2.2 Integrity

This section addresses the question of who can request which changes.

7.2.2.1 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.

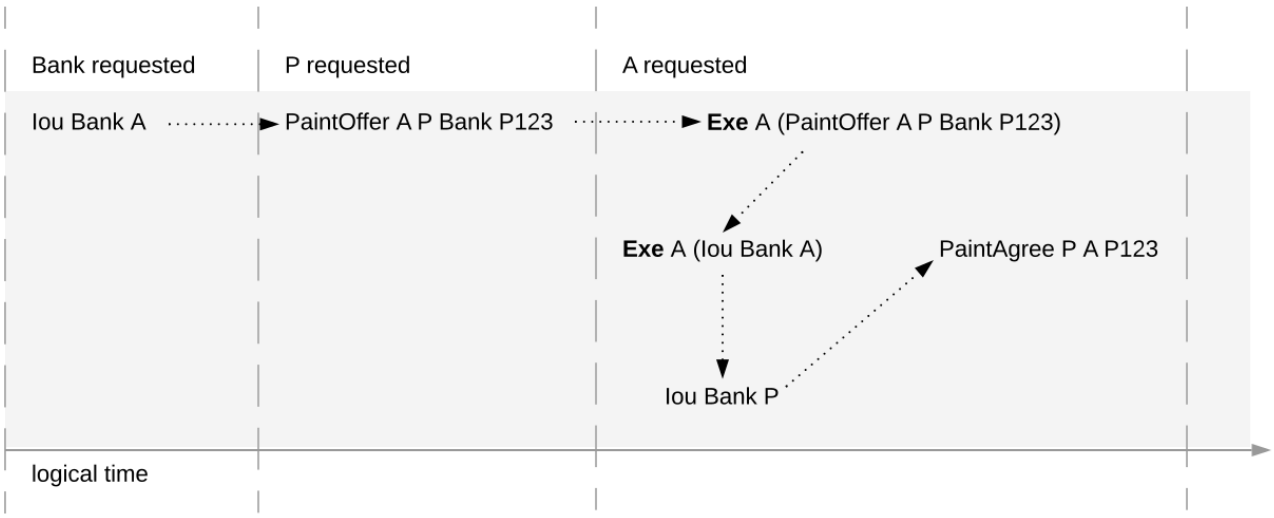
Only the last of these conditions depends on the party (or parties) requesting the change; the other two are general.

7.2.2.2 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*.

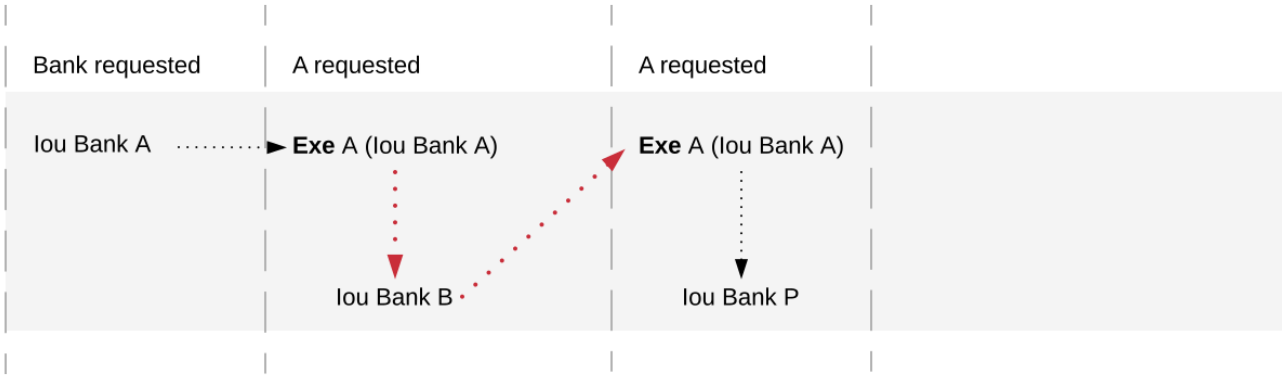
Contract consistency

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*.

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

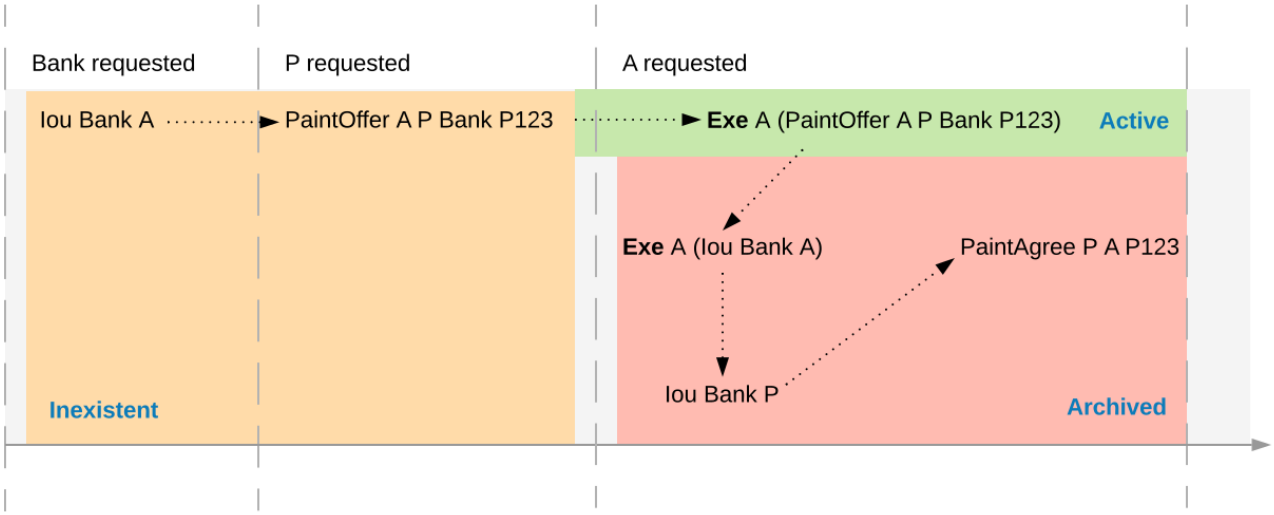


Fig. 7: Activeness of the *PaintOffer* contract

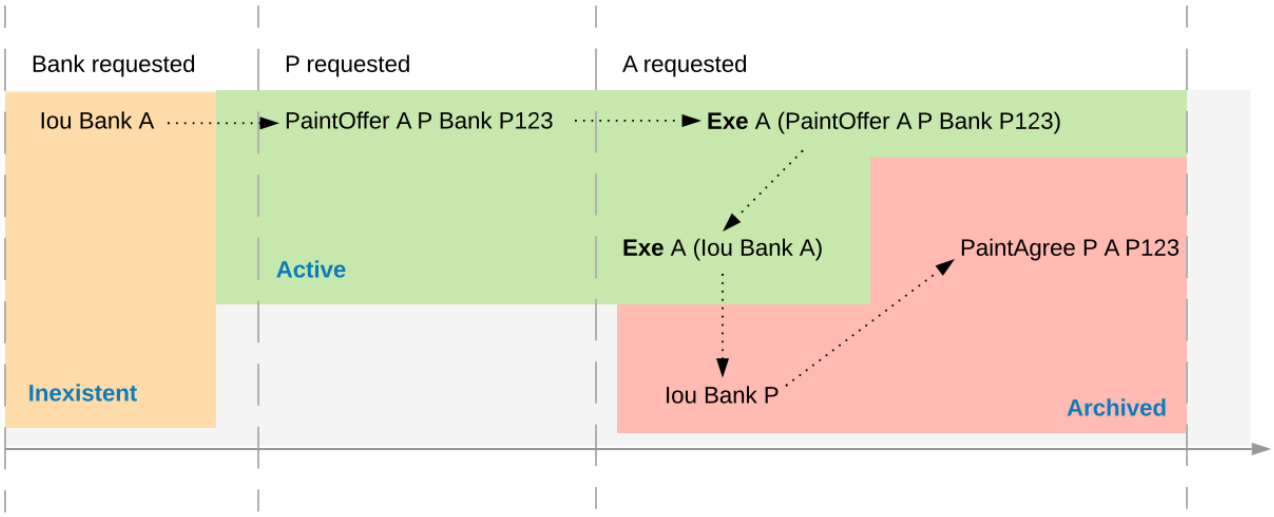


Fig. 8: Activeness of the *Iou Bank A* contract

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.

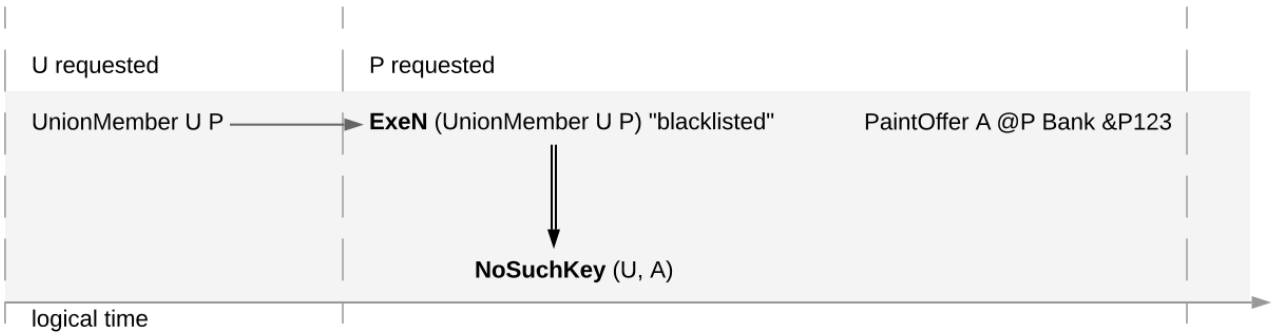
Definition key consistency A ledger is **consistent for a key** *k* if for every action *act* on *k*, the key state *s* before *act* satisfies

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)*.

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*.



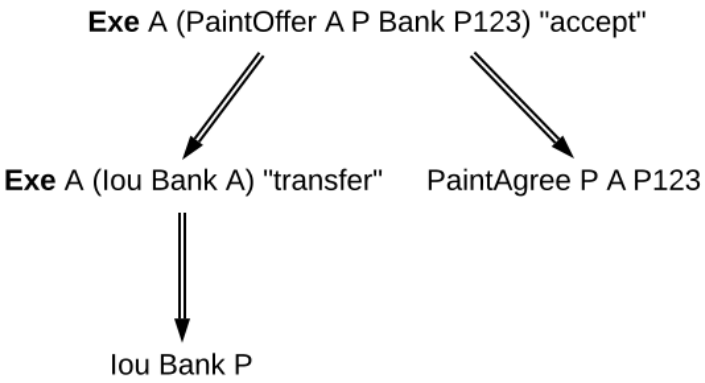
Key consistency extends to actions, transactions and lists of transactions just like the other consistency notions.

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.

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.

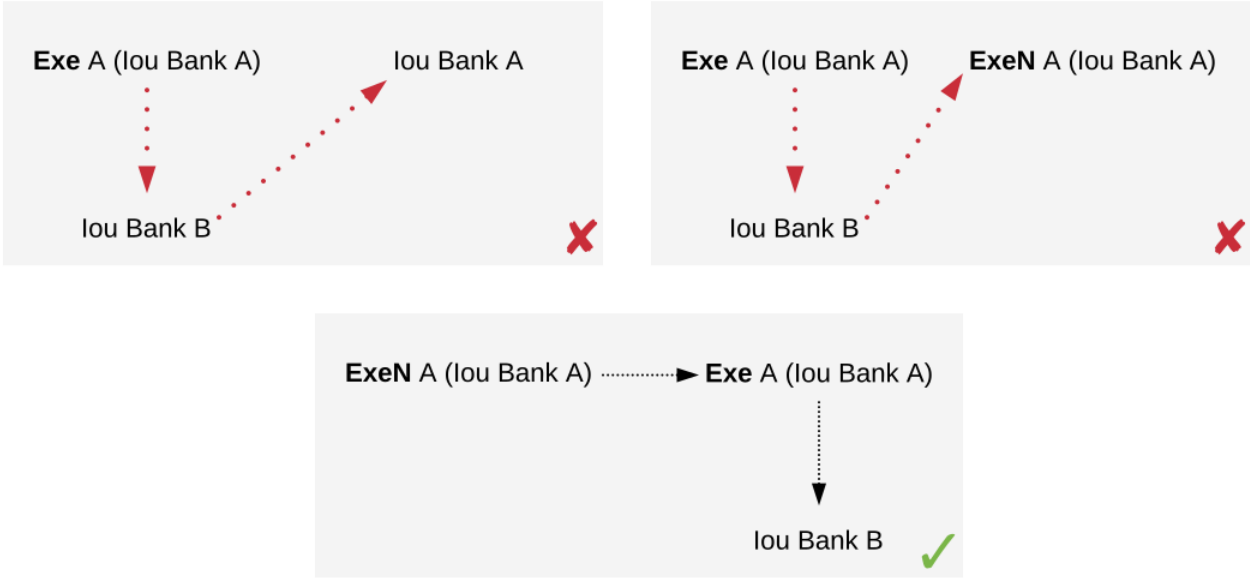


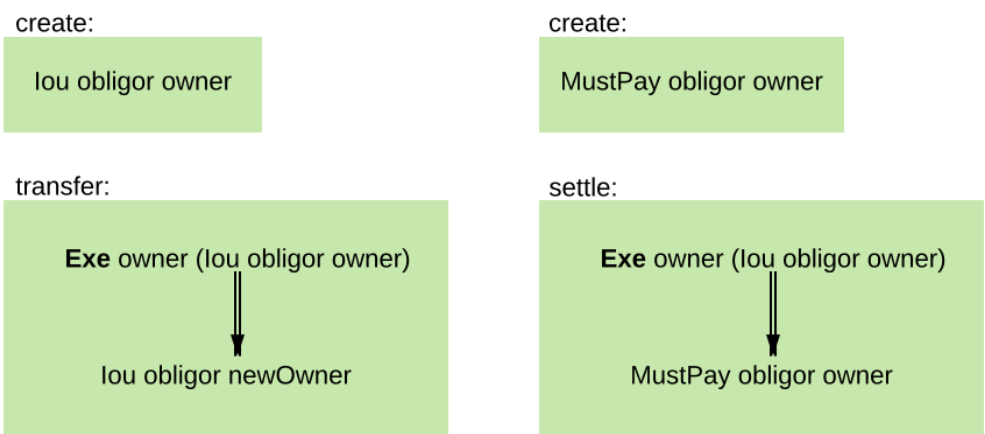
Fig. 9: The first two transactions violate the conditions of internal consistency. The first transaction creates the *Iou* after exercising it consumingly, violating both conditions. The second transaction contains a (non-consuming) exercise on the *Iou* after a consuming one, violating the second condition. The last transaction is internally consistent.

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.

7.2.2.3 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.

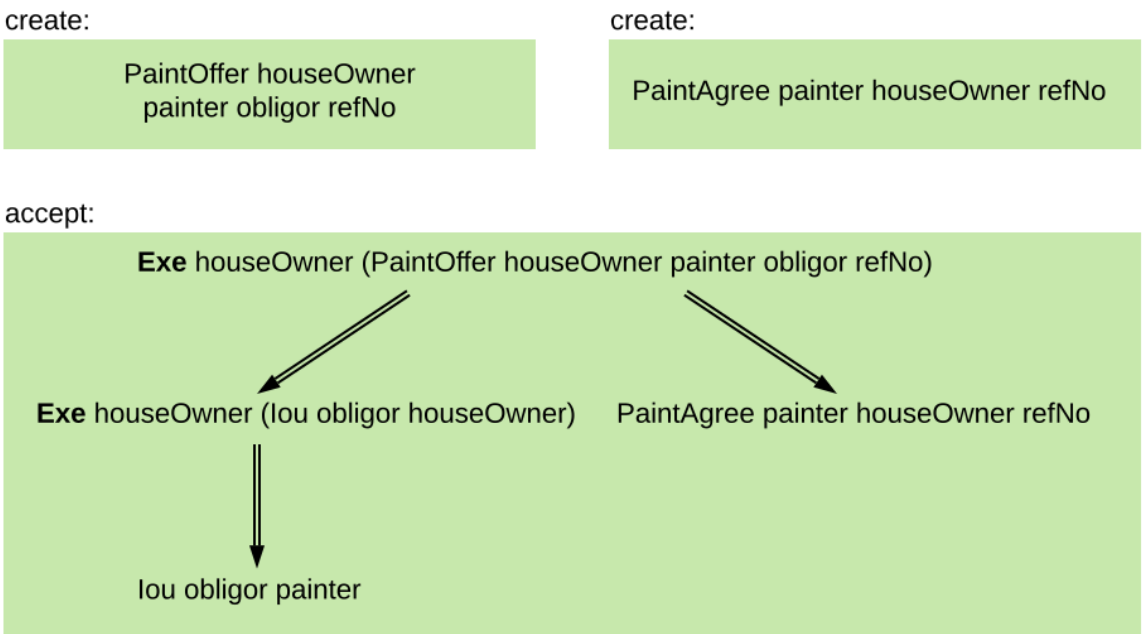
For example, the set of allowed actions on IOU contracts could be described as follows.



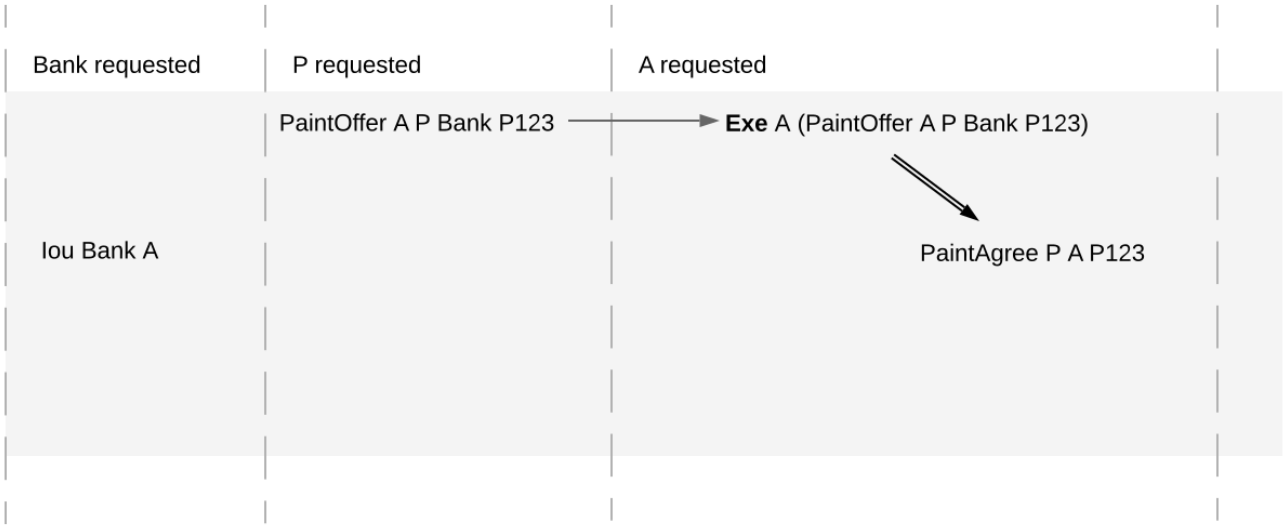
The boxes in the image are templates in the sense that the contract parameters in a box (such as obligor or owner) can be instantiated by arbitrary values of the appropriate type. To facilitate understanding, each box includes a label describing the intuitive purpose of the corresponding set of actions. 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.



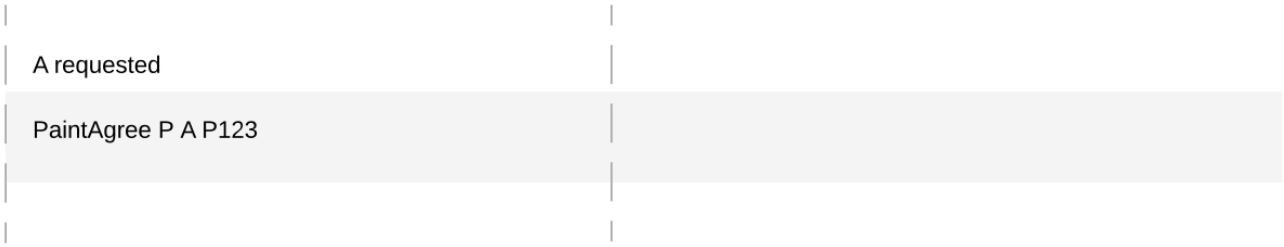
The problem with example where Alice changes the offer's outcome to avoid transferring the money now becomes apparent.



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.

7.2.2.4 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:

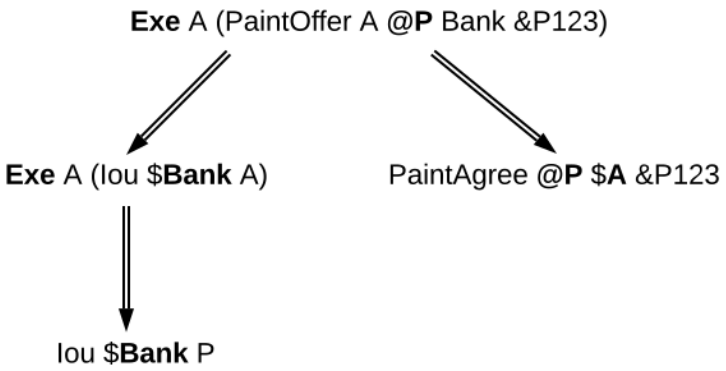
1. A non-empty set of **signatories**, the parties bound by the contract.
2. An optional **agreement text** associated with the contract, specifying the off-ledger, real-world obligations of the signatories.
3. If the contract is associated with a key, a non-empty set of **maintainers**, the parties 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:

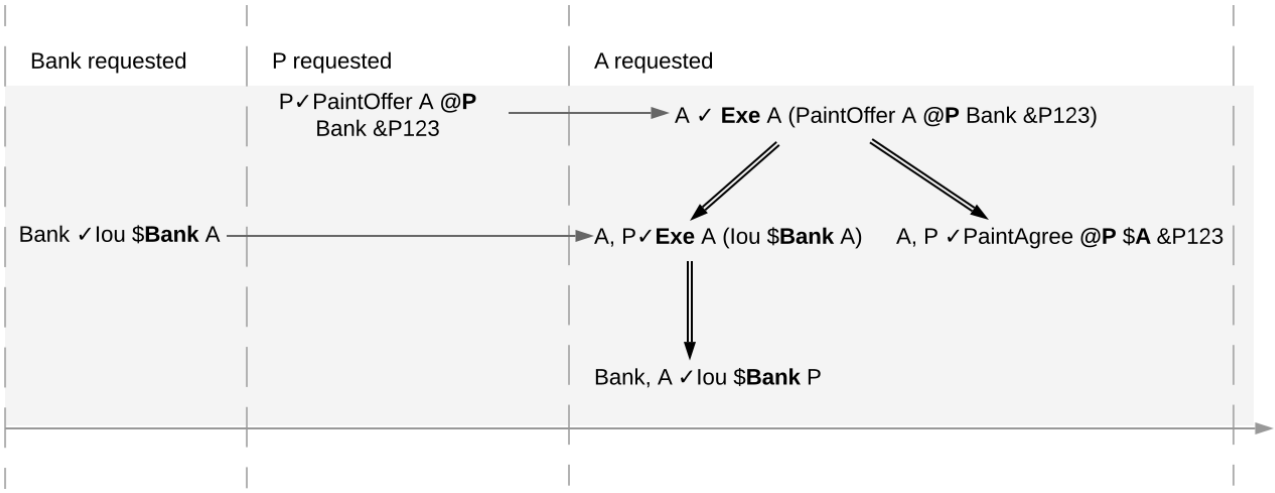
1. the required authorizers of a **Create** action on a contract *c* are the signatories of *c*.
2. the required authorizers of an **Exercise** or a **Fetch** action are its actors.
3. the required authorizers of a **NoSuchKey** assertion are the maintainers of the key.

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.

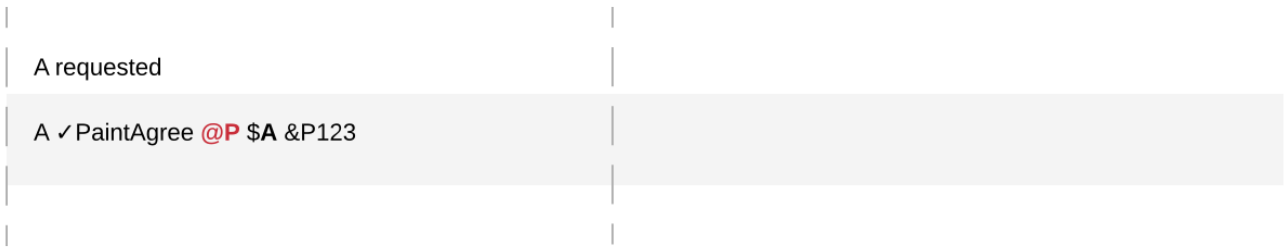
In the visualizations below, $\Pi \checkmark$ *act* denotes that the parties Π authorize the action *act*. The resulting authorizations are shown below.



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.

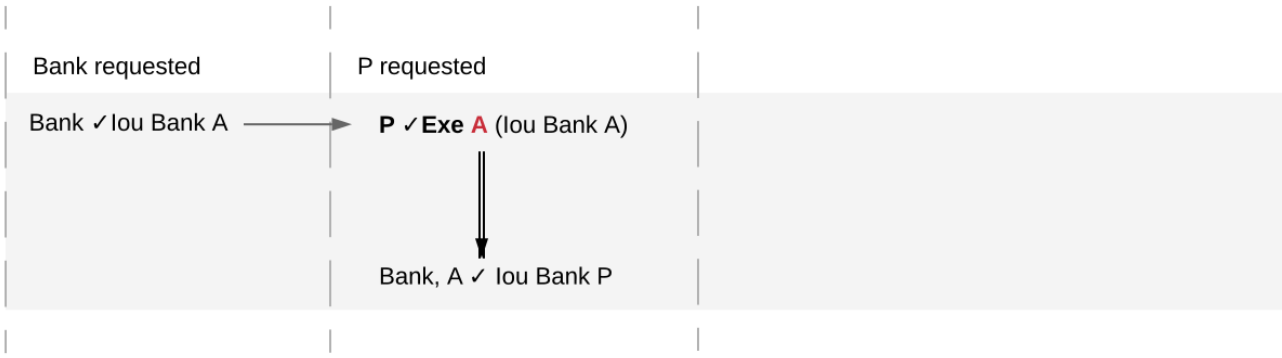
Similarly, the intuitively problematic examples are prohibited by our authorization criterion. In the first example, Alice forced the painter to paint her house. The authorizations for the example are shown below.



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*.

7.2.2.5 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.

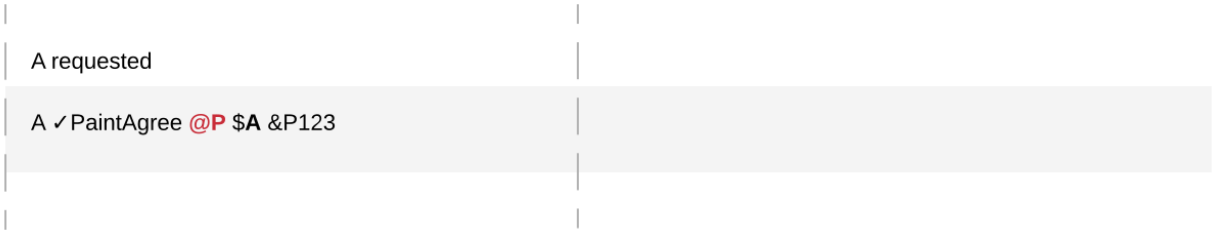
Second, every contract on a Daml ledger can simultaneously model both:

> 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.

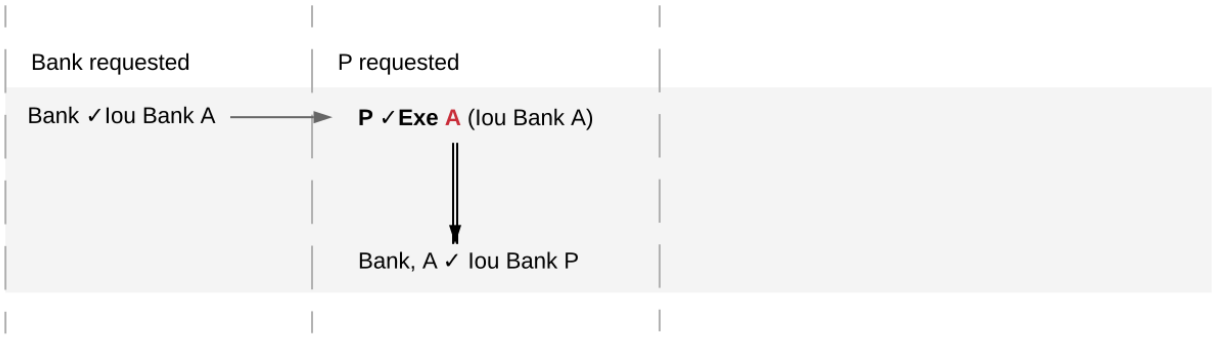
Third, in Daml ledgers, as in the real world, one person's rights are another person's obligations. For example, *A*'s right to accept the *PaintOffer* is *P*'s obligation to paint her house in case she accepts. In Daml ledgers, a party's rights according to a contract model are the exercise actions 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.

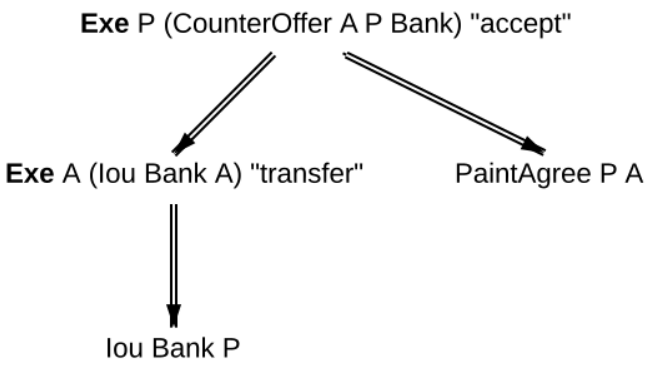
1. **Obligations need consent**. Daml ledgers follow the offer-acceptance pattern of the contract law, and thus ensures that all ledger contracts are formed voluntarily. For example, the following ledger is not valid.



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.

7.2.3 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.

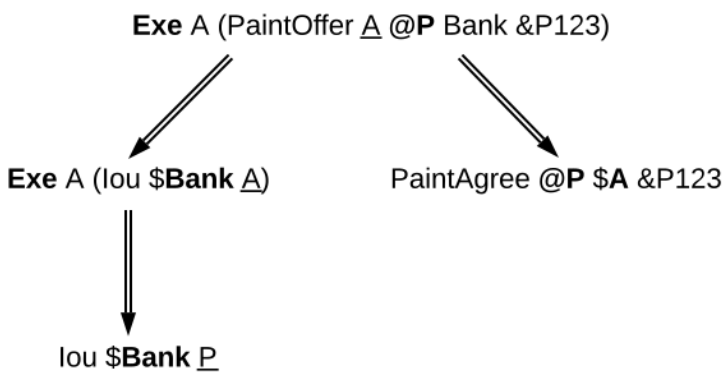
To make this more precise, a stakeholder concept is needed.

7.2.3.1 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.



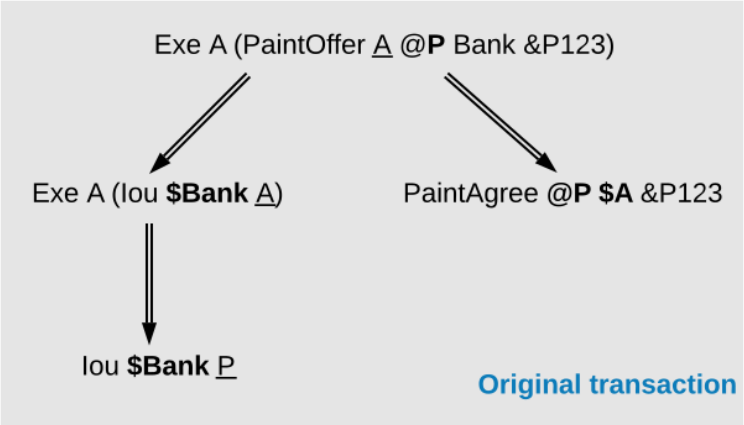
7.2.3.2 Choice Observers

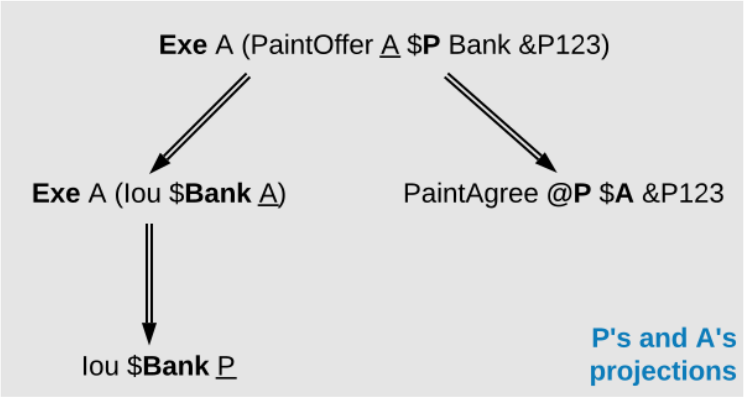
In addition to contract observers, the contract model can also specify **choice observers** on individual **Exercise** actions. Choice observers get to see a specific exercise on a contract, and to view its consequences. Choice observers are not considered stakeholders of the contract, they only affect the set of informees on an action, for the purposes of projection (see below).

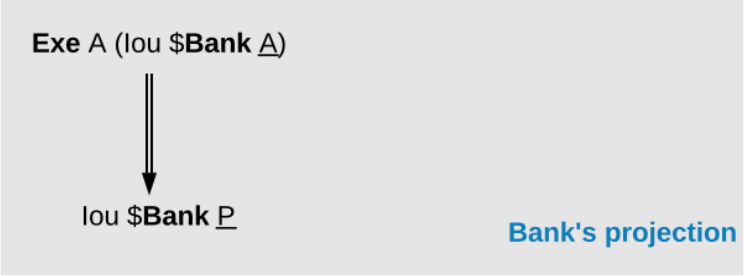
7.2.3.3 Projections

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 observers - 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 subtransaction 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.

7.2.3.4 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.

7.2.3.5 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.

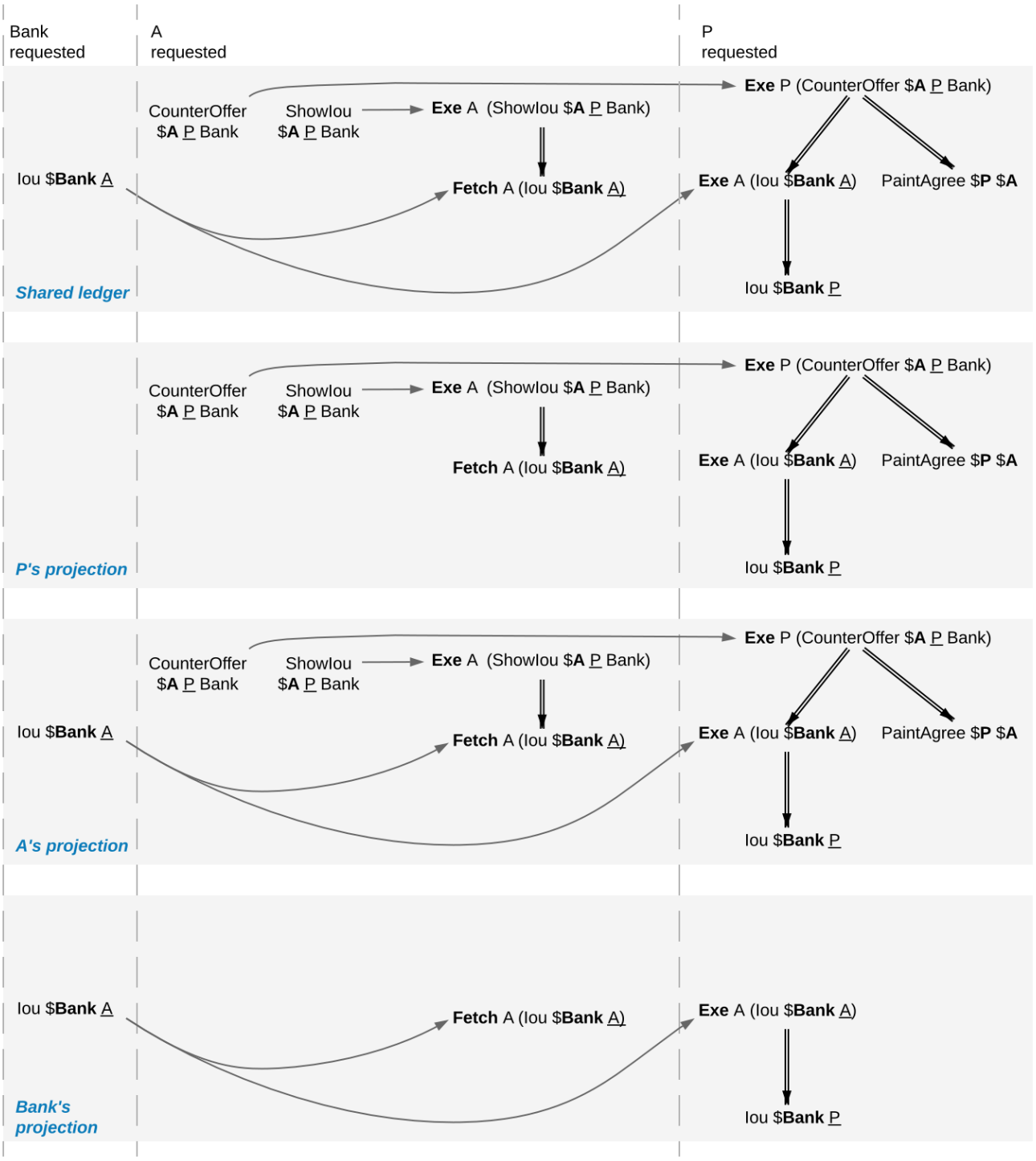
In the example of *ledger projections of the paint offer*, the exercise on the *PaintOffer* is visible to both the painter and Alice. As a consequence, the exercise on the *Iou Bank A* is visible to the painter, and the

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.

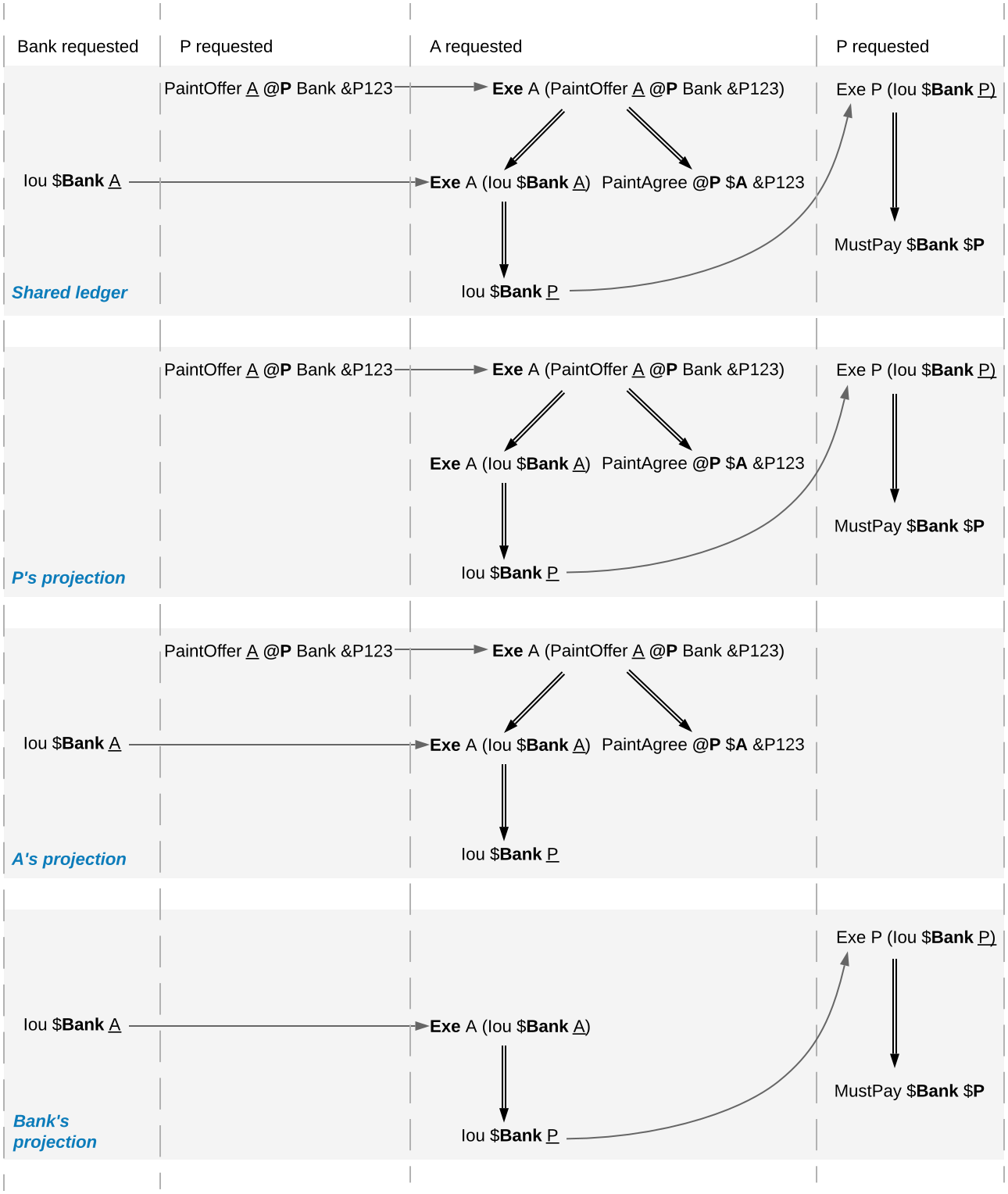
Showing contracts to non-stakeholders through ledger projections is called **divulgence**. Divulgence is a deliberate choice in the design of Daml ledgers. In the paint offer example, the only proper way to accept the offer is to transfer the money from Alice to the painter. Conceptually, at the instant 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.

7.2.4 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

    controller owner can
      Transfer
        : ContractId Iou
        with newOwner : Party
        do create Iou with obligor; owner = newOwner

    controller owner can
      Settle
        : ContractId MustPay
        do create MustPay with obligor; owner
```

In this example, the owner is automatically made an observer on the contract, as the `Transfer` and `Settle` choices use the `controller owner can` syntax.

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.

7.2.5 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.

7.2.5.1 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.

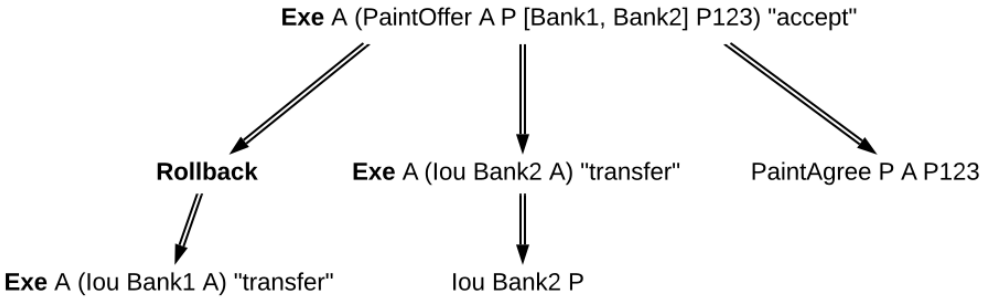
To support this in our ledger model, we need to modify the transaction structure to indicate which subtransactions were rolled back. We do this by introducing **rollback nodes** in the transaction. Each rollback node contains a rolled back subtransaction. Rollback nodes are not considered ledger actions.

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'
```

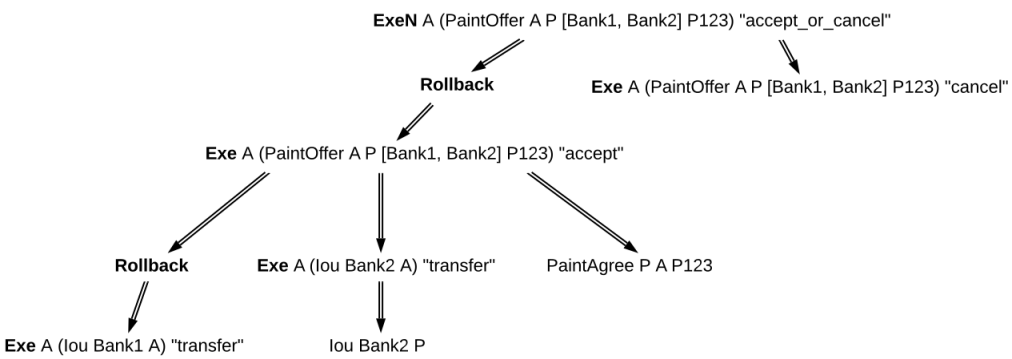
Note that *Action* and *Kind* have the same definition as before, but since *Transaction* may now contain rollback nodes, this means that an *Exercise* action may have a rollback node as one of its consequences.

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.



7.2.5.2 Consistency

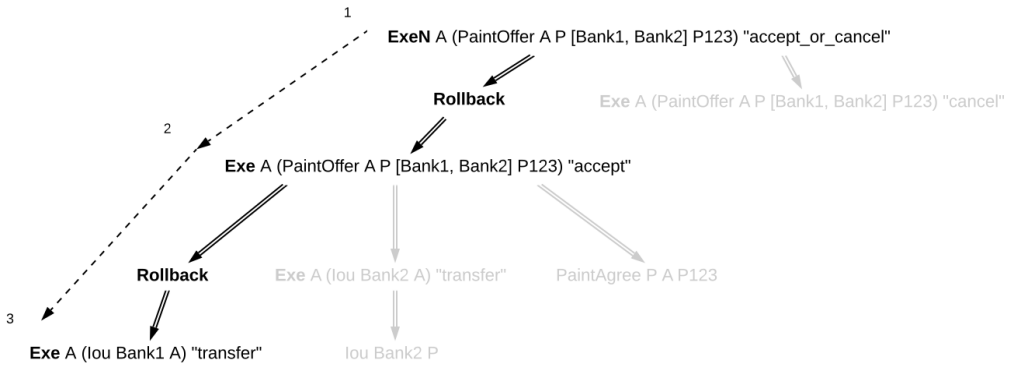
In the previous section on *consistency*, we defined a before-after relation on ledger actions. This notion needs to be revised in the presence of rollback nodes. It is no longer enough to perform a preorder traversal of the transaction tree, because the actions under a rollback node cannot affect actions 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.

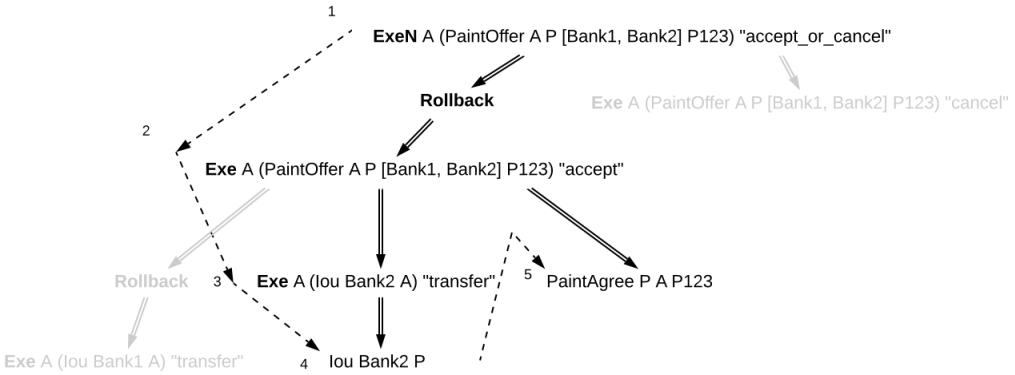
So, we now define the before-after relation as a partial order, rather than a total order, on all the actions of a transaction. This relation is defined as follows: *act1* comes before *act2* (equivalently, *act2* comes after *act1*) if and only if *act1* appears before *act2* in a preorder traversal 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*.

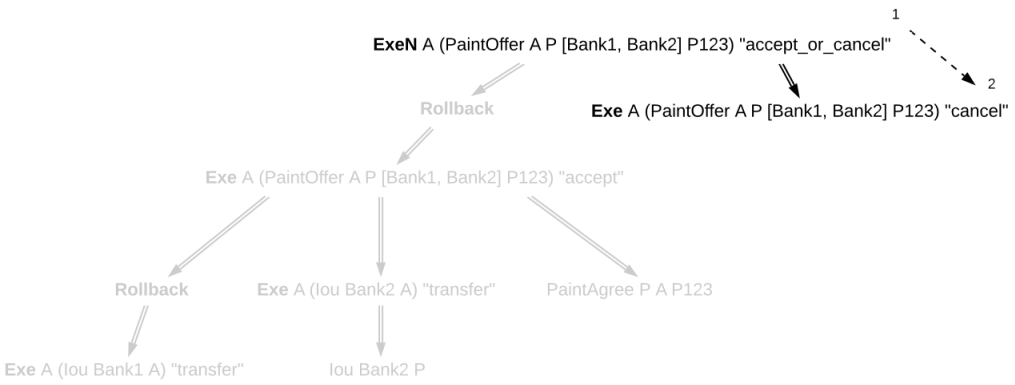
In the example above, neither consuming exercise comes after the other. They are part of separate continuities, so they don't introduce inconsistency. Here are three continuities implied by the before-after relation. The first:



The second:



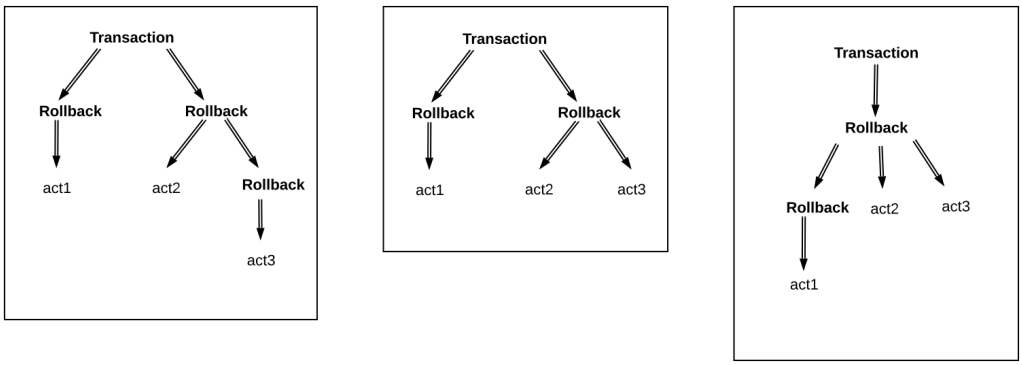
And the third:



As you can see, in each of these continuities, no contract was consumed twice.

7.2.5.3 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*):

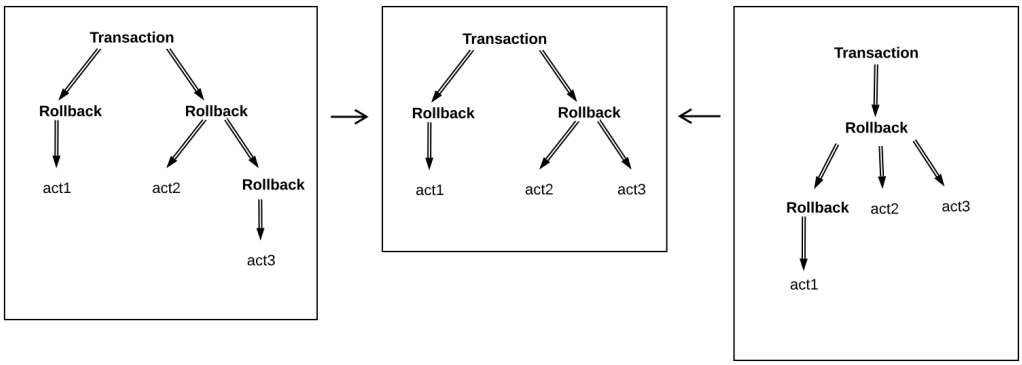


Because of this, these three transactions are equivalent. More generally, two transactions are equivalent if:

> 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.

7.2.5.4 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.

7.2.5.5 Privacy

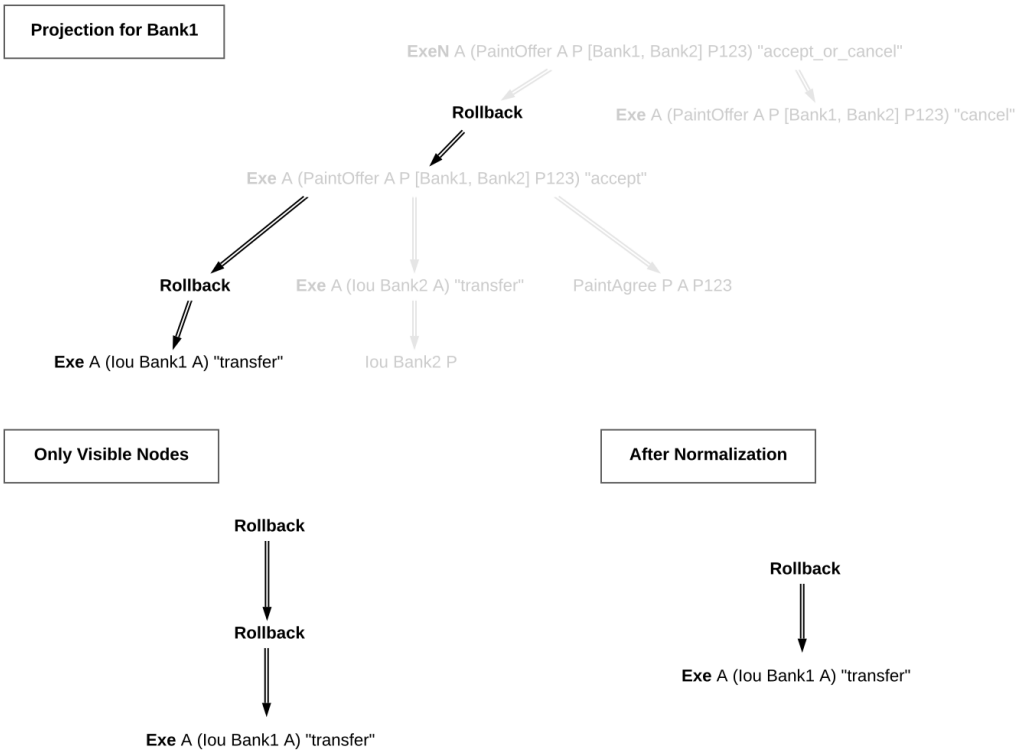
Rollback nodes also have an interesting effect on the notion of privacy in the ledger model. When projecting a transaction for a party *p*, it's necessary to preserve some of the rollback structure of the transaction, even if *p* does not have the right to observe every action under it. For example, 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.

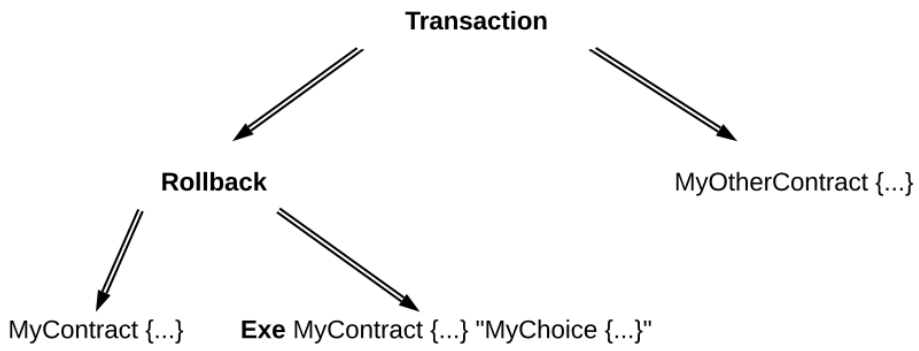
7.2.5.6 Relation to Daml Exceptions

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.

7.3 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 the *ledger's topology* influences the unspecified part of the behavior.

7.3.1 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.

7.3.1.1 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. In global state topologies, 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. In partitioned state topologies, 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). If the ledger has a *global state topology*, the new identifier will generally be allocated and vetted by the operator of the writer node(s). For example, in the *replicated committer topology*, the committers can jointly decide on whether to approve the provisioning, and which identifier to return. If they refuse to provision the identifier, the method call fails.

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`. This does not yet guarantee that `Alice_123` can now submit a command creating a new contract with `Bob_456` as an observer. In general, `Alice_123` will be able to do this in a ledger with a *global state topology*. In such ledgers, the nodes holding the physical shared ledger typically also maintain a central directory of all parties in the system. However, such a directory may not exist for a ledger with a *partitioned topology*. In fact, in such a ledger, the participants `P1` and `P2` might not have a way to communicate to each other, or might not even be aware of each other's existence.

For diagnostics, the ledger also 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.

7.3.1.2 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*.

7.3.1.3 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.

A *global state topology* might 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. . Ledgers with *partitioned topologies* in general 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.

7.3.2 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.

7.3.2.1 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.

7.3.2.2 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.

7.3.2.3 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. For example, in *global state topologies*, every *trust domain* that controls how commits are appended to the shared ledger must execute Daml code. Thus, the operators of these trust domains will in general be allowed to vet the code before they execute it. 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. In a replicated ledger, the vetting might require consent from all or a quorum of replicas. 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 *partitioned topologies*, individual trust domains store only parts of the ledger. Thus, they only need to approve packages whose templates are used in the ledger part visible to them. For example, in Daml on R3 Corda, participants only need to approve code for the contracts in their parties' projections. If non-validating Corda notaries are used, they do not need to vet code. If validating Corda notaries are used, they can also choose which code to vet. 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`.

7.3.2.4 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.

7.4 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`.

7.4.1 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.

7.4.2 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.

7.4.3 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.

7.4.4 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.

7.4.5 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.

7.5 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*.

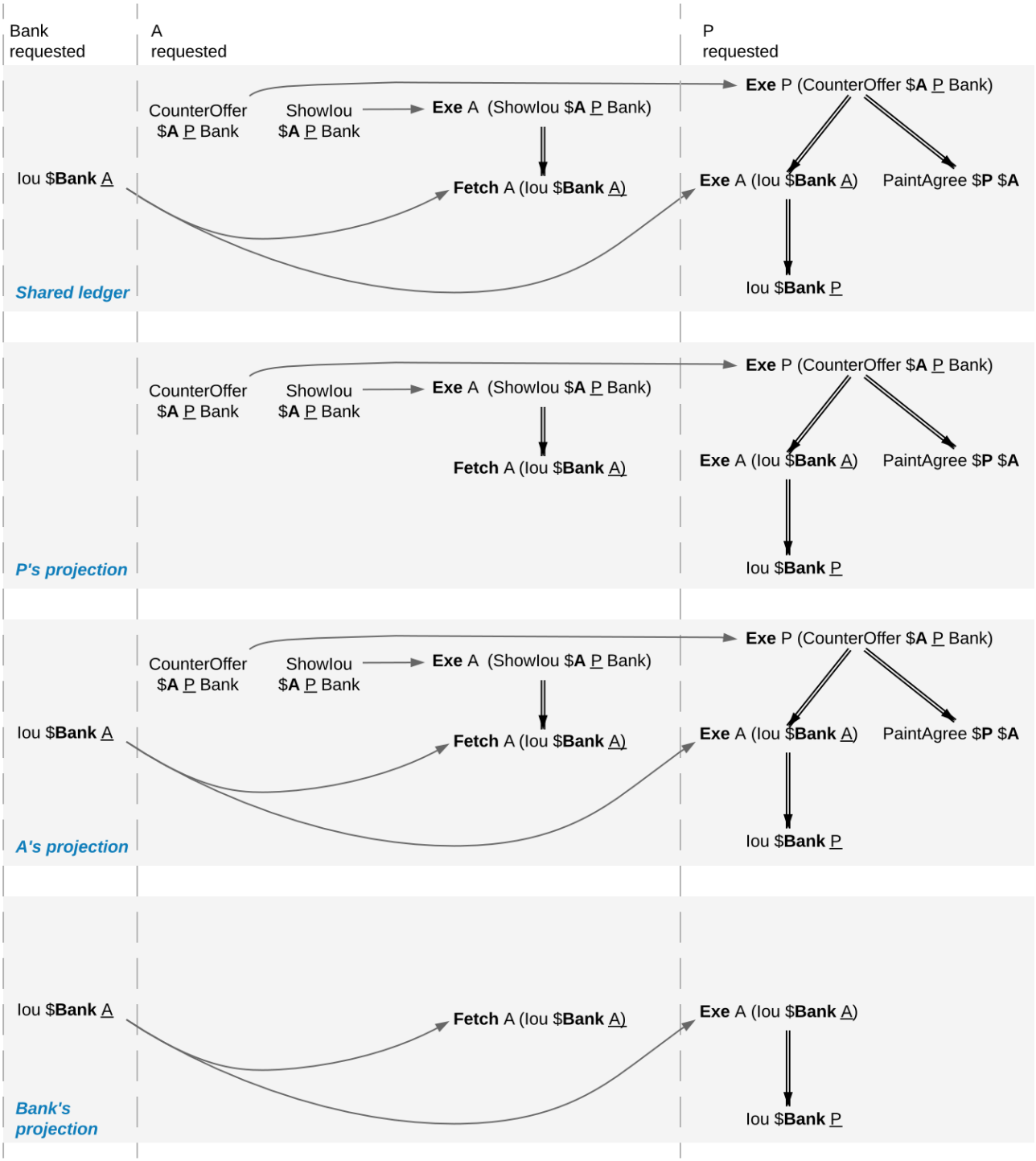
The presentation assumes that you are familiar with the following concepts:

The *Ledger API*
The *Daml Ledger Model*

7.5.1 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.



7.5.1.1 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.

7.5.1.2 Signatories of a contract and stakeholder actors see usages after the creation and before the archival.

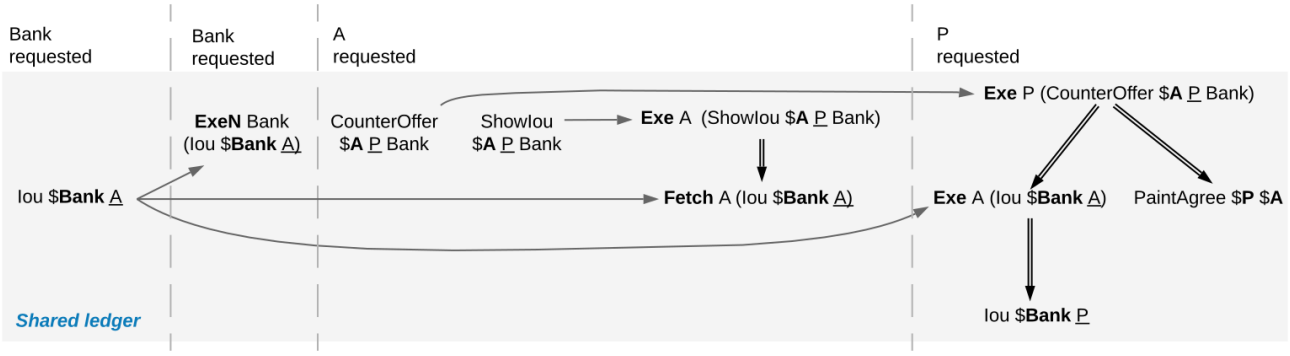
The *Fetch A (Iou Bank A)* action comes after the creation of the *Iou Bank A* and before its archival, for both Alice and the Bank, because the Bank is a signatory of the *Iou Bank A* contract and Alice is a stakeholder of the *Iou Bank A* contract and an actor on the **Fetch** action.

7.5.1.3 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*.

7.5.1.4 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.

7.5.1.5 Out-of-band causality is not respected.

The following examples assume that Alice splits up her commit into two as follows:

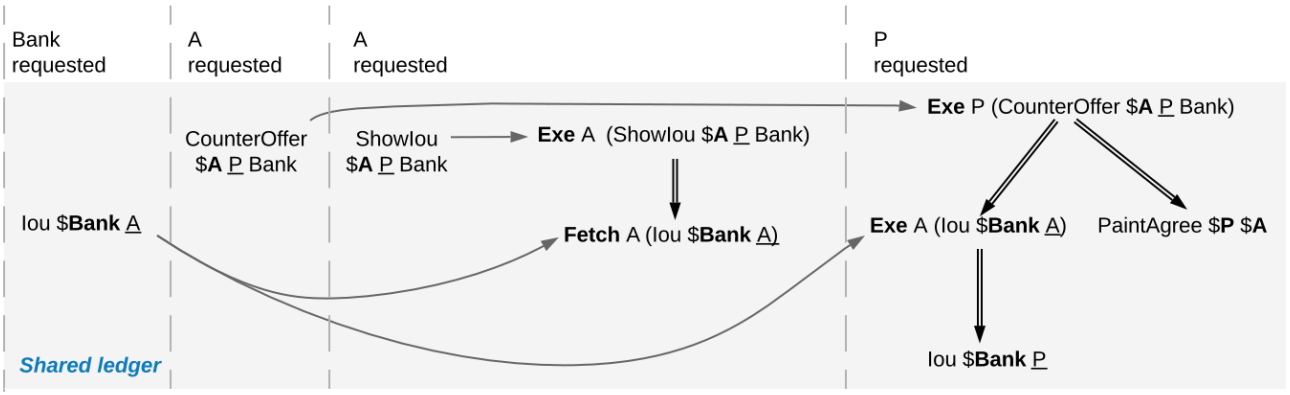


Fig. 10: Counteroffer workflow with four commits.

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.

7.5.1.6 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.

7.5.1.7 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.

7.5.2 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 or-

der 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.

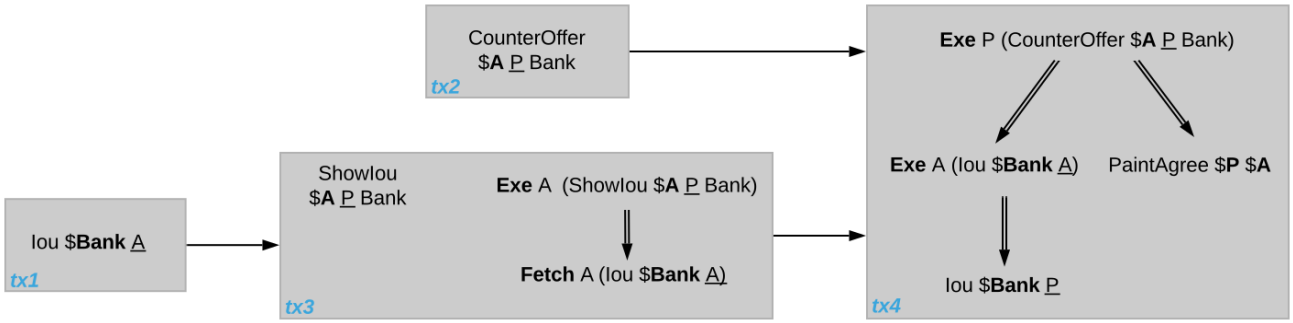


Fig. 11: Causality graph for the *counteroffer workflow with four commits*.

7.5.2.1 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.

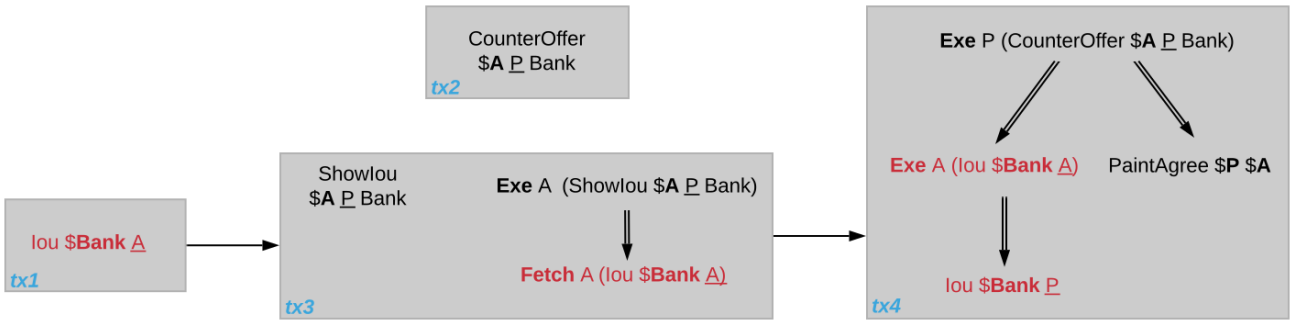
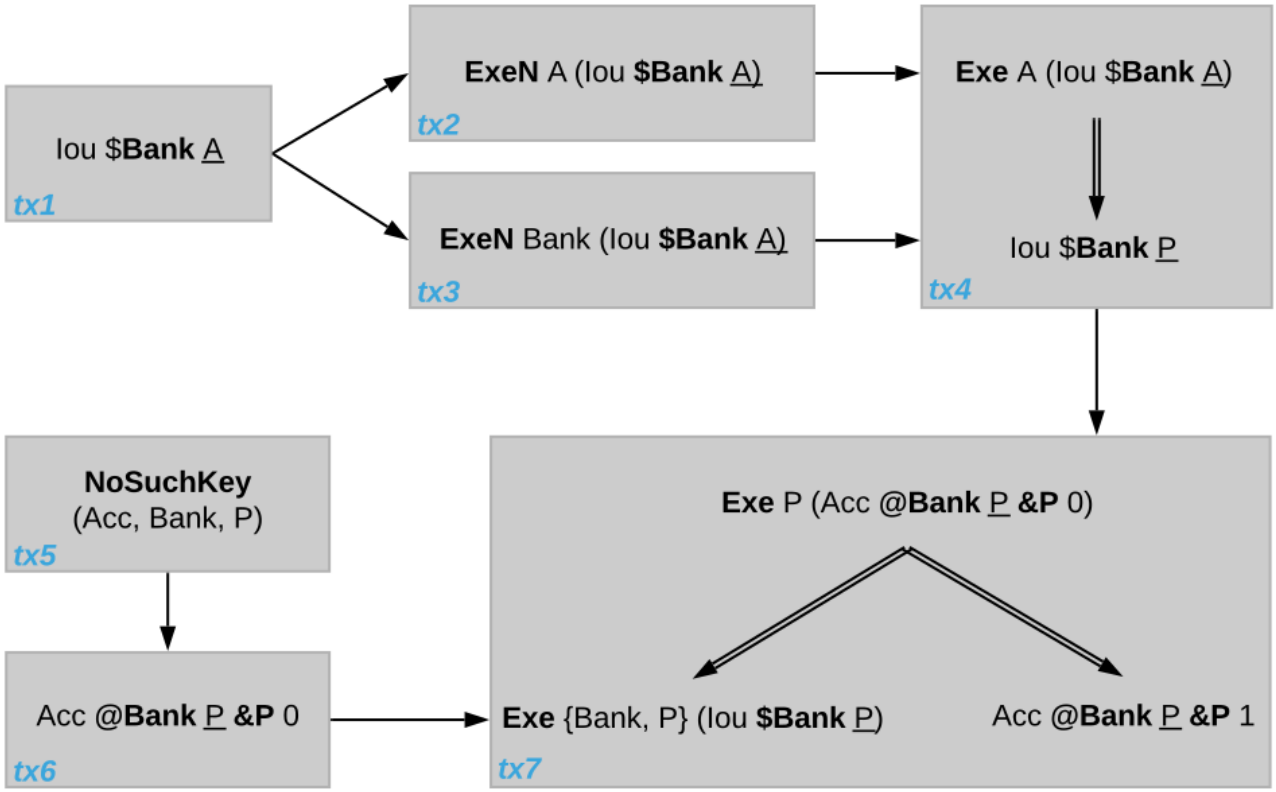


Fig. 12: Minimal consistent causality graph for the highlighted actions.

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.

7.5.2.2 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 $reduce(G_L)$.

7.5.3 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.

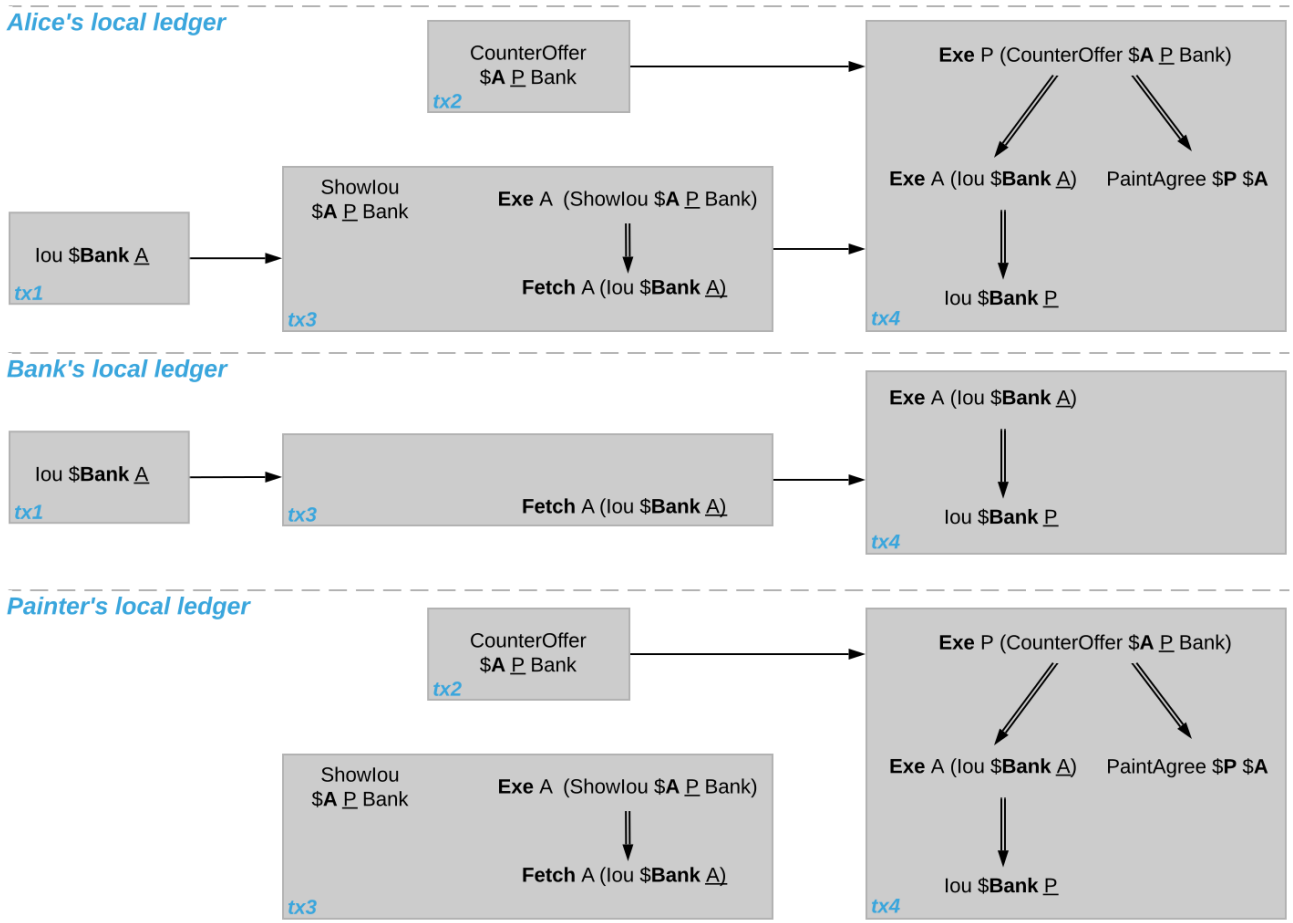


Fig. 13: Projections of the *split counteroffer causality graph*.

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*.

7.5.3.1 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.

7.5.3.2 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.

Chapter 8

Examples

Chapter 9

Early Access Features

9.1 Extractor

The Extractor is currently an *Early Access Feature in Labs status*.

9.1.1 Introduction

You can use the Extractor to extract contract data for a single party from a Ledger node into a PostgreSQL database.

It is useful for:

Application developers to access data on the ledger, observe the evolution of data, and debug their applications
Business analysts to analyze ledger data and create reports
Support teams to debug any problems that happen in production

Using the Extractor, you can:

Take a full snapshot of the ledger (from the start of the ledger to the current latest transaction)
Take a partial snapshot of the ledger (between specific offsets)
Extract historical data and then stream indefinitely (either from the start of the ledger or from a specific offset)

9.1.2 Setting up

Prerequisites:

A PostgreSQL database that is reachable from the machine the Extractor runs on. Use PostgreSQL version 9.4 or later to have JSONB type support that is used in the Extractor.
We recommend using an empty database to avoid schema and table collisions. To see which tables to expect, see *Output format*.
A running Sandbox or Ledger Node as the source of data.
You've *installed the SDK*.

Once you have the prerequisites, you can start the Extractor like this:

```
$ daml extractor --help
```

9.1.3 Trying it out

This example extracts:

> all contract data from the beginning of the ledger to the current latest transaction
> for the party `Scrooge_McDuck`
> from a Ledger node or Sandbox running on host `192.168.1.12` on port `6865`
> to PostgreSQL instance running on localhost
> identified by the user `postgres` without a password set
> into a database called `daml_export`

```
$ daml extractor postgresql --user postgres --connecturl
↪jdbc:postgresql:daml_export --party Scrooge_McDuck -h 192.168.1.12 -
↪p 6865 --to head
```

This terminates after reaching the transaction which was the latest at the time the Extractor started streaming.

To run the Extractor indefinitely, and thus keeping the database up to date as new transactions arrive on the ledger, omit the `--to head` parameter to fall back to the default streaming-indefinitely approach, or state explicitly by using the `--to follow` parameter.

9.1.4 Running the Extractor

The basic command to run the Extractor is:

```
$ daml extractor [options]
```

For what options to use, see the next sections.

9.1.5 Connecting the Extractor to a ledger

To connect to the Sandbox, provide separate address and port parameters. For example, `--host 10.1.1.10 --port 6865`, or in short form `-h 10.1.1.168 -p 6865`.

The default host is `localhost` and the default port is `6865`, so you don't need to pass those.

To connect to a Ledger node, you might have to provide SSL certificates. The options for doing this are shown in the output of the `--help` command.

9.1.6 Connecting to your database

As usual for a Java application, the database connection is handled by the well-known JDBC API, so you need to provide:

> a JDBC connection URL
> a username
> an optional password

For more on the connection URL, visit https://jdbc.postgresql.org/documentation/80/connect.html.

This example connects to a PostgreSQL instance running on `localhost` on the default port, with a user `postgres` which does not have a password set, and a database called `daml_export`. This is a typical setup on a developer machine with a default PostgreSQL install

```
$ daml extractor postgres --connecturl jdbc:postgresql:daml_export --user
↪postgres --party [party]
```

This example connects to a database on host `192.168.1.12`, listening on port `5432`. The database is called `daml_export`, and the user and password used for authentication are `daml_exporter` and `ExamplePassword`

```
$ daml extractor postgres --connecturl jdbc:postgresql://192.168.1.12:5432/
↪daml_export --user daml_exporter --password ExamplePassword --party
↪[party]
```

9.1.7 Authorize Extractor

If you are running Extractor against a Ledger API server that verifies authorization, you must provide the access token when you start it.

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 Extractor by storing it in a file and provide the path to it using the `--access-token-file` command line option.

Both in the case in which the token cannot be read from the provided path or if the Ledger API reports an authorization error (for example due to token expiration), Extractor will keep trying to read and use it and report the error via logging. This retry mechanism allows expired token to be overwritten with valid ones and keep Extractor going from where it left off.

9.1.8 Full list of options

To see the full list of options, run the `--help` command, which gives the following output:

```
Usage: extractor [prettyprint|postgresql] [options]

Command: prettyprint [options]
Pretty print contract template and transaction data to stdout.
  --width <value>          How wide to allow a pretty-printed value to
↪become before wrapping.
                           Optional, default is 120.
  --height <value>         How tall to allow each pretty-printed output to
↪become before
                           it is truncated with a `...`.
                           Optional, default is 1000.

Command: postgresql [options]
Extract data into a PostgreSQL database.
  --connecturl <value>     Connection url for the `org.postgresql.Driver`
↪driver. For examples,
                           visit https://jdbc.postgresql.org/documentation/
↪80/connect.html
  --user <value>           The database user on whose behalf the
↪connection is being made.
  --password <value>       The user's password. Optional.

Common options:
  -h, --ledger-host <h>    The address of the Ledger host. Default is 127.
↪0.0.1
```

(continues on next page)

(continued from previous page)

```
  -p, --ledger-port <p>    The port of the Ledger host. Default is 6865.
  --ledger-api-inbound-message-size-max <bytes>
                           Maximum message size from the ledger API.
↪Default is 52428800 (50MiB).
  --party <value>          The party or parties whose contract data should
↪be extracted.
                         Specify multiple parties separated by a comma, e.
↪g. Foo,Bar
  -t, --templates <module1>:<entity1>,<module2>:<entity2>...
                           The list of templates to subscribe for.
↪Optional, defaults to all ledger templates.
  --from <value>           The transaction offset (exclusive) for the
↪snapshot start position.
                           Must not be greater than the current latest
↪transaction offset.
                           Optional, defaults to the beginning of the
↪ledger.
                           Currently, only the integer-based Sandbox
↪offsets are supported.
  --to <value>             The transaction offset (inclusive) for the
↪snapshot end position.
                           Use "head" to use the latest transaction offset
↪at the time
                           the extraction first started, or "follow" to
↪stream indefinitely.
                           Must not be greater than the current latest
↪offset.
                           Optional, defaults to "follow".
  --help                   Prints this usage text.

TLS configuration:
  --pem <value>            TLS: The pem file to be used as the private key.
  --crt <value>            TLS: The crt file to be used as the cert chain.
                           Required if any other TLS parameters are set.
  --cacrt <value>          TLS: The crt file to be used as the trusted
↪root CA.

Authorization:
  --access-token-file <value>
                           provide the path from which the access token
↪will be read, required if the Ledger API server verifies authorization,
↪no default
```

Some options are tied to a specific subcommand, like `--connecturl` only makes sense for the `postgresql`, while others are general, like `--party`.

9.1.9 Output format

To understand the format that Extractor outputs into a PostgreSQL database, you need to understand how the ledger stores data.

The Daml Ledger is composed of transactions, which contain events. Events can represent:

creation of contracts (create event), or
exercise of a choice on a contract (exercise event).

A contract on the ledger is either active (created, but not yet archived), or archived. The relationships between transactions and contracts are captured in the database: all contracts have pointers (foreign keys) to the transaction in which they were created, and archived contracts have pointers to the transaction in which they were archived.

9.1.10 Transactions

Transactions are stored in the `transaction table` in the `public` schema, with the following structure

```
CREATE TABLE transaction
  (transaction_id TEXT PRIMARY KEY NOT NULL
  ,seq BIGSERIAL UNIQUE NOT NULL
  ,workflow_id TEXT
  ,effective_at TIMESTAMP NOT NULL
  ,extracted_at TIMESTAMP DEFAULT NOW()
  ,ledger_offset TEXT NOT NULL
  );
```

transaction_id: The transaction ID, as appears on the ledger. This is the primary key of the table.
transaction_id, effective_at, workflow_id, ledger_offset: These columns are the properties of the transaction on the ledger. For more information, see the specification.
seq: Transaction IDs should be treated as arbitrary text values: you can't rely on them for ordering transactions in the database. However, transactions appear on the Ledger API transaction stream in the same order as they were accepted on the ledger. You can use this to work around the arbitrary nature of the transaction IDs, which is the purpose of the `seq` field: it gives you a total ordering of the transactions, as they happened from the perspective of the ledger. Be aware that `seq` is not the exact index of the given transaction on the ledger. Due to the privacy model of Daml Ledgers, the transaction stream won't deliver a transaction which doesn't concern the party which is subscribed. The transaction with `seq` of 100 might be the 1000th transaction on the ledger; in the other 900, the transactions contained only events which mustn't be seen by you.
extracted_at: The `extracted_at` field means the date the transaction row and its events were inserted into the database. When extracting historical data, this field will point to a possibly much later time than `effective_at`.

9.1.11 Contracts

Create events and contracts that are created in those events are stored in the `contract` table in the `public` schema, with the following structure

```
CREATE TABLE contract
  (event_id TEXT PRIMARY KEY NOT NULL
  ,archived_by_event_id TEXT DEFAULT NULL
  ,contract_id TEXT NOT NULL
  ,transaction_id TEXT NOT NULL
  ,archived_by_transaction_id TEXT DEFAULT NULL
```

(continues on next page)

(continued from previous page)

```
  ,is_root_event BOOLEAN NOT NULL
  ,package_id TEXT NOT NULL
  ,template TEXT NOT NULL
  ,create_arguments JSONB NOT NULL
  ,witness_parties JSONB NOT NULL
  );
```

event_id, contract_id, create_arguments, witness_parties: These fields are the properties of the corresponding `CreatedEvent` class in a transaction. For more information, see the specification.
package_id, template: The fields `package_id` and `template` are the exploded version of the `template_id` property of the ledger event.
transaction_id: The `transaction_id` field refers to the transaction in which the contract was created.
archived_by_event_id, archived_by_transaction_id: These fields will contain the event id and the transaction id in which the contract was archived once the archival happens.
is_root_event: Indicates whether the event in which the contract was created was a root event of the corresponding transaction.

Every contract is placed into the same table, with the contract parameters put into a single column in a JSON-encoded format. This is similar to what you would expect from a document store, like MongoDB. For more information on the JSON format, see the *later section*.

9.1.12 Exercises

Exercise events are stored in the `exercise` table in the `public` schema, with the following structure:

```
CREATE TABLE
  exercise
  (event_id TEXT PRIMARY KEY NOT NULL
  ,transaction_id TEXT NOT NULL
  ,is_root_event BOOLEAN NOT NULL
  ,contract_id TEXT NOT NULL
  ,package_id TEXT NOT NULL
  ,template TEXT NOT NULL
  ,contract_creating_event_id TEXT NOT NULL
  ,choice TEXT NOT NULL
  ,choice_argument JSONB NOT NULL
  ,acting_parties JSONB NOT NULL
  ,consuming BOOLEAN NOT NULL
  ,witness_parties JSONB NOT NULL
  ,child_event_ids JSONB NOT NULL
  );
```

package_id, template: The fields `package_id` and `template` are the exploded version of the `template_id` property of the ledger event.
is_root_event: Indicates whether the event in which the contract was created was a root event of the corresponding transaction.
transaction_id: The `transaction_id` field refers to the transaction in which the contract was created.
The other columns are properties of the `ExercisedEvent` class in a transaction. For more

information, see the specification.

9.1.13 JSON format

Extractor stores create and choice arguments using the *Daml-LF JSON Encoding*. The parameters of the JSON schema are instantiated as follows in Extractor:

```
encodeDecimalAsString: true
encodeInt64AsString: false
```

9.1.14 Examples of output

The following examples show you what output you should expect. The Sandbox has already run the scenarios of a Daml model that created two transactions: one creating a `Main:RightOfUseOffer` and one accepting it, thus archiving the original contract and creating a new `Main:RightOfUseAgreement` contract. We also added a new offer manually.

This is how the `transaction` table looks after extracting data from the ledger:

transaction_id	seq	workflow_id	effective_at	extracted_at	ledger_offset
scenario-transaction-0	1	scenario-workflow-0	1970-01-01 01:00:00	2019-03-08 15:14:18.481316	1
scenario-transaction-1	2	scenario-workflow-1	1970-01-01 01:00:00	2019-03-08 15:14:18.521912	2
2	3	ae267813270cb865	1970-01-01 01:00:00	2019-03-08 15:14:18.560584	3

You can see that the transactions which were part of the scenarios have the format `scenario-transaction-{n}`, while the transaction created manually is a simple number. This is why the `seq` field is needed for ordering. In this output, the `ledger_offset` field has the same values as the `seq` field, but you should expect similarly arbitrary values there as for transaction IDs, so better rely on the `seq` field for ordering.

This is how the `contract` table looks:

event_id	archived_by_event_id	contract_id	transaction_id	archived_by_transaction_id	is_root_ev...	package_id	template	create_arguments	witness_parties
#2:0	NULL	#2:0	2	NULL	TRUE	528d2184c218aa9b0b960db7882cd5abc6dba83078aab1dc8e9dbe6860a5a548	Main:RightOfUseOffer	{"tenant": "Scrooge_McDuck", "addr...	["Betina_Beakley"]
#scenario-transaction-0:0:0	NULL	#0:0	scenario-transaction-0	NULL	TRUE	528d2184c218aa9b0b960db7882cd5abc6dba83078aab1dc8e9dbe6860a5a548	Main:RightOfUseOffer	{"tenant": "Betina_Beakley", "address": "McDuck Man...	["Betina_Beakley"]
#scenario-transaction-1:1:1	NULL	#1:1	scenario-transaction-1	NULL	FALSE	528d2184c218aa9b0b960db7882cd5abc6dba83078aab1dc8e9dbe6860a5a548	Main:RightOfUseAgreement	{"tenant": "Betina_Beakley", "address": "McDuck Man...	["Betina_Beakley"]

You can see that the `archived_by_transaction_id` and `archived_by_event_id` fields of contract `#0:0` is not empty, thus this contract is archived. These fields of contracts `#1:1` and `#2:0` are `NULL` s, which mean they are active contracts, not yet archived.

This is how the `exercise` table looks:

event_id	transaction_id	is_root_ev...	contract_id	package_id	template	contract_creating_event_id	choice	choice_argument	acting_parties	consuming	witness_parties	child_event_ids
#scenario-transaction-1:1:0	scenario-transaction-1	TRUE	#0:0	528d2184c218aa9b0b960db7882cd5abc6dba83...	Main:RightOfUseOffer	#0:0	Accept	{}	["Betina_Beakley"]	TRUE	["Betina_Beakley"]	["#scenario-transaction-1:1:1"]

You can see that there was one exercise `Accept` on contract `#0:0`, which was the consuming choice mentioned above.

9.1.15 Dealing with schema evolution

When updating packages, you can end up with multiple versions of the same package in the system.

Let's say you have a template called `My.Company.Finance.Account`:

```
module My.Company.Finance.Account where

template Account
  with
    provider: Party
    accountId: Text
    owner: Party
    observers: [Party]
  where
    [...]
```

This is built into a package with a resulting hash `6021727fe0822d688ddd545997476d530023b222d02f191`

Later you add a new field, `displayName`:

```
module My.Company.Finance.Account where

template Account
  with
    provider: Party
    accountId: Text
    owner: Party
    observers: [Party]
    displayName: Text
  where
    [...]
```

The hash of the new package with the update is `1239d1c5df140425f01a5112325d2e4edf2b7ace223f8c1d`

There are contracts of first version of the template which were created before the new field is added, and there are contracts of the new version which were created since. Let's say you have one instance of each:

```
{
  "owner":"Bob",
  "provider":"Bob",
  "accountId":"6021-5678",
  "observers":[
      "Alice"
  ]
}
```

and:

```
{
  "owner":"Bob",
  "provider":"Bob",
  "accountId":"1239-4321",
  "observers":[
      "Alice"
  ],
```

(continues on next page)

(continued from previous page)

```
  "displayName":"Personal"
}
```

They will look like this when extracted:

id	seq	event_id	transaction_id	archived_by_transaction_id	package_id	template	contract
#3:0	3	#3:0	3	NULL	1239d1c5df140425f01a5112325d2e4edf2b7ace223f8c1d2ebebe76a8ececfe	My.Company.Finance.Account	{"owner": "Bob", "provider": "Bob", "accountId": "1239-4321", "observers": ["Alice"], "displayName": "Personal"}
#4:0	4	#4:0	4	NULL	6021727fe0822d688ddd545997476d530023b222d02f1919567bd82b205a5ce3	My.Company.Finance.Account	{"owner": "Bob", "provider": "Bob", "accountId": "6021-5678", "observers": ["Alice"]}

To have a consistent view of the two versions with a default value `NULL` for the missing field of instances of older versions, you can create a view which contains all `Account` rows:

```
CREATE VIEW account_view AS
SELECT
   create_arguments->>'owner' AS owner
  ,create_arguments->>'provider' AS provider
  ,create_arguments->>'accountId' AS accountId
  ,create_arguments->>'displayName' AS displayName
  ,create_arguments->'observers' AS observers
FROM
  contract
WHERE
  package_id =
↪'1239d1c5df140425f01a5112325d2e4edf2b7ace223f8c1d2ebebe76a8ececfe'
  AND
  template = 'My.Company.Finance.Account'
UNION
SELECT
   create_arguments->>'owner' AS owner
  ,create_arguments->>'provider' AS provider
  ,create_arguments->>'accountId' AS accountId
  ,NULL as displayName
  ,create_arguments->'observers' AS observers
FROM
  contract
WHERE
  package_id =
↪'6021727fe0822d688ddd545997476d530023b222d02f1919567bd82b205a5ce3'
  AND
  template = 'My.Company.Finance.Account';
```

Then, `account_view will` contain both contracts:

owner	provider	accountid	displayname	observers
Bob	Bob	1239-4321	Personal	["Alice"]
Bob	Bob	6021-5678	NULL	["Alice"]

9.1.16 Logging

By default, the Extractor logs to stderr, with INFO verbose level. To change the level, use the `-DLOGLEVEL=[level]` option, e.g. `-DLOGLEVEL=TRACE`.

You can supply your own logback configuration file via the standard method: https://logback.qos.ch/manual/configuration.html

9.1.17 Continuity

When you terminate the Extractor and restart it, it will continue from where it left off. This happens because, when running, it saves its state into the `state` table in the `public` schema of the database. When started, it reads the contents of this table. If there's a saved state from a previous run, it restarts from where it left off. There's no need to explicitly specify anything, this is done automatically.

DO NOT modify content of the `state` table. Doing so can result in the Extractor not being able to continue running against the database. If that happens, you must delete all data from the database and start again.

If you try to restart the Extractor against the same database but with different configuration, you will get an error message indicating which parameter is incompatible with the already exported data. This happens when the settings are incompatible: for example, if previously contract data for the party `Alice` was extracted, and now you want to extract for the party `Bob`.

The only parameters that you can change between two sessions running against the same database are the connection parameters to both the ledger and the database. Both could have moved to different addresses, and the fact that it's still the same Ledger will be validated by using the Ledger ID (which is saved when the Extractor started its work the first time).

9.1.18 Fault tolerance

Once the Extractor connects to the Ledger Node and the database and creates the table structure from the fetched Daml packages, it wraps the transaction stream in a restart logic with an exponential backoff. This results in the Extractor not terminating even when the transaction stream is aborted for some reason (the ledger node is down, there's a network partition, etc.).

Once the connection is back, it continues the stream from where it left off. If it can't reach the node on the host/port pair the Extractor was started with, you need to manually stop it and restart with the updated address.

Transactions on the ledger are inserted into PostgreSQL as atomic SQL transactions. This means either the whole transaction is inserted or nothing, so you can't end up with inconsistent data in the database.

9.1.19 Troubleshooting

9.1.19.1 Can't connect to the Ledger Node

If the Extractor can't connect to the Ledger node on startup, you'll see a message like this in the logs, and the Extractor will terminate:

```
16:47:51.208 ERROR c.d.e.Main$@[akka.actor.default-dispatcher-7] - FAILURE:
io.grpc.StatusRuntimeException: UNAVAILABLE: io exception.
Exiting...
```

To fix this, make sure the Ledger node is available from where you're running the Extractor.

9.1.19.2 Can't connect to the database

If the database isn't available before the transaction stream is started, the Extractor will terminate, and you'll see the error from the JDBC driver in the logs:

```
17:19:12.071 ERROR c.d.e.Main$@[kka.actor.default-dispatcher-5] - FAILURE:
org.postgresql.util.PSQLException: FATAL: database "192.153.1.23:daml_
↪export" does not exist.
Exiting…
```

To fix this, make sure make sure the database exists and is available from where you're running the Extractor, the username and password your using are correct, and you have the credentials to connect to the database from the network address where the you're running the Extractor.

If the database connection is broken while the transaction stream was already running, you'll see a similar message in the logs, but in this case it will be repeated: as explained in the *Fault tolerance* section, the transaction stream will be restarted with an exponential backoff, giving the database, network or any other trouble resource to get back into shape. Once everything's back in order, the stream will continue without any need for manual intervention.

9.2 Daml Integration Kit

The Daml integration kit is currently an *Early Access Feature in Labs status*. It comprises the components needed to build your own *Daml Drivers*.

9.2.1 Ledger API Test Tool

The Ledger API Test Tool is a command line tool for testing the correctness of implementations of the *Ledger API*, i.e. Daml ledgers. For example, it will show you if there are consistency or conformance problem with your implementation.

Its intended audience are developers of Daml ledgers, who are using the Daml Ledger Implementation Kit to develop a Daml ledger on top of their distributed-ledger or database of choice.

Use this tool to verify if your Ledger API endpoint conforms to the *DA Ledger Model*.

9.2.1.1 Downloading the tool

Download the Ledger API Test Tool from Maven and save it as `ledger-api-test-tool.jar` in your current directory.

9.2.1.2 Running the tool against a custom Ledger API endpoint

Run this command to test your Ledger API endpoint exposed at host `<host>` and at a port `<port>`:

```
$ java -jar ledger-api-test-tool.jar <host>:<port>
```

For example:

```
$ java -jar ledger-api-test-tool.jar localhost:6865
```

The tool will upload the required DARs to the ledger, and then run all tests.

If any test embedded in the tool fails, it will print out details of the failure for further debugging.

9.2.1.3 Exploring options the tool provides

Run the tool with `--help` flag to obtain the list of options the tool provides:

```
$ java -jar ledger-api-test-tool.jar --help
```

Selecting tests to run

Running the tool without any argument runs only the *default tests*.

Those include all tests that are known to be safe to be run concurrently as part of a single run.

Tests that either change the global state of the ledger (e.g. configuration management) or are designed to stress the implementation need to be explicitly included using the available command line options.

Use the following command line flags to select which tests to run:

> `--list`: print all available test suites to the console, shows if they are run by default
> `--list-all`: print all available tests to the console, shows if they are run by default
> `--include`: only run the tests that match the argument
> `--exclude`: do not run the tests that match the argument
> `--perf-tests`: list performance tests to run; cannot be combined with normal tests
> `--skip-dar-upload`: skip upload of DAR files into ledger. DAR files should be uploaded manually before the tests.

Include and exclude are matched as prefixes, e.g. `--exclude=SemanticTests` will exclude all tests whose name starts with `SemanticTests`. Test names always start with their suite name followed by a colon, so the test suite names shown by `--list` can be useful for coarse-grained inclusion/exclusion.

Both `--include` and `--exclude` (and `--perf-tests`) can be specified multiple times and/or provide comma-separated lists, i.e. all of these are equivalent:

```
--include=a,b,c
--include=a --include=b --include=c
--include=a,b --include=c
```

The logic is always to first select included tests, then remove from that the excluded ones, i.e. include directives never override a corresponding exclude directive.

If no `--include` flag is given, all of the tests are included. You cannot run performance and non-performance tests in the same invocation. `--exclude` is ignored when running performance tests, and the program will stop if it detects that both `--perf-tests` and `--include` have been specified.

Examples (hitting a single participant at `localhost:6865`):

Listing 1: Only run `TestA`

```
$ java -jar ledger-api-test-tool.jar --include TestA localhost:6865
```

Listing 2: Run all tests, but not `TestB`

```
$ java -jar ledger-api-test-tool.jar --exclude TestB localhost:6865
```

Listing 3: Run all tests

```
$ java -jar ledger-api-test-tool.jar localhost:6865
```

Listing 4: Run all tests, but not `TestC`

```
$ java -jar ledger-api-test-tool.jar --exclude TestC
```

Performance tests

The available performance tests allow to establish the performance envelope of the ledger under test (a term borrowed from aeronautics), which offers an indication of the amount of the parameters under which a ledger implementation is supposed to perform.

Those tests include tail latency, throughput and maximum size of a single transaction. You can run the tool with the `--list` option to see a list of available test suites that includes individual performance envelope test cases. You can mix and match those tests to produce a test suite tailored to match the expected performance envelope of a given ledger implementation using a specific hardware setup.

For example, the following will verify that the ledger under test can have a tail latency of one second when processing twenty pings, perform twenty pings per seconds and being able to process a transaction one megabyte in size:

```
$ java -jar ledger-api-test-tool.jar \
  --perf-tests=PerformanceEnvelope.Latency.1000ms \
  --perf-tests=PerformanceEnvelope.Throughput.TwentyOPS \
  --perf-tests=PerformanceEnvelope.TransactionSize.1000KB \
  localhost:6865
```

Note: A ping is a collective name for two templates used to evaluate the performance envelope. Each of the two templates, Ping and Pong , have a single choice allowing the controller to create an instance of the complementary template, directed to the original sender.

The test run will also produce a short summary of statistics which is printed to standard output by default but that can be written to a specific file path using the `--perf-tests-report` command line option.

9.2.1.4 Try out the Ledger API Test Tool against Daml Sandbox

If you wanted to test out the tool, you can run it against *Daml Sandbox*. To do this:

```
$ java -jar ledger-api-test-tool.jar --extract
$ daml sandbox *.dar
$ java -jar ledger-api-test-tool.jar localhost:6865
```

This should always succeed, as the Sandbox is tested to correctly implement the Ledger API. This is useful if you do not have yet a custom Ledger API endpoint.

9.2.1.5 Using the tool with a known-to-be-faulty Ledger API implementation

Use flag `--must-fail` if you expect one or more or the scenario tests to fail. If enabled, the tool will return the success exit code when at least one test fails, and it will return a failure exit code when all tests succeed:

```
java -jar ledger-api-test-tool.jar --must-fail localhost:6865
```

This is useful during development of a Daml ledger implementation, when tool needs to be used against a known-to-be-faulty implementation (e.g. in CI). It will still print information about failed tests.

9.2.1.6 Tuning the testing behaviour of the tool

Use the command line option `--timeout-scale-factor` to tune timeouts applied by the tool.

Set `--timeout-scale-factor` to a floating point value higher than 1.0 to make the tool wait longer for expected events coming from the Daml ledger implementation under test. Conversely use values smaller than 1.0 to make it wait shorter.

9.2.1.7 Accomodating different ledger clock intervals

Use the command line option `--ledger-clock-granularity` to indicate the maximum interval at which the ledger's clock will increment.

If running on a ledger where ledger time increments in a time period greater than 10s, set `--ledger-clock-granularity` to a value higher than 10000 (10,000ms). Tests that are sensitive to the ledger clock will then wait for a corresponding longer period of time to ensure completion of operations, avoiding timeouts and premature failures. The command deduplication test suite is particularly sensitive to this value.

9.2.1.8 Verbose output

Use the command line option `--verbose` to print full stack traces on failures.

9.2.1.9 Concurrent test runs

To minimize concurrent runs of tests, `--concurrent-test-runs` can be set to 1 or 2. The default value is the number of processors available.

Note that certain tests, known to be possibly interfering with others (e.g. configuration management), are always run sequentially and as the last tests in a run.

9.2.1.10 Retired tests

A few tests can be retired over time as they could be deemed not providing the necessary signal to a developer or operator that an integration correctly implements the Daml Ledger API. Those test will nominally be kept in the test suite for a time to prevent unwanted breakages of existing CI pipelines. They will however not be run and they will eventually be removed. You are advised to remove any explicit reference to those tests while they are in their deprecation period.

Retired tests are not listed when using `--list` or `--list-all` but can be included in a run using `--include`. In this case, nothing will be run and the test report will mention that the test has been retired and skipped.

Daml Applications run on Daml Ledgers. A Daml Ledger is a server serving the *Ledger API* as per the semantics defined in the *Daml Ledger Model* and the Daml-LF specification.

The Daml integration kit helps third-party ledger developers to implement a Daml Ledger on top of their distributed ledger or database of choice.

We provide the resources in the kit, which include guides to

Daml Integration Kit status and roadmap
Implementing your own Daml Ledger
Deploying a Daml Ledger
Testing a Daml Ledger
Benchmarking a Daml Ledger

Using these guides, you can focus on your own distributed-ledger or database and reuse our Daml Ledger server and Daml interpreter code for implementing the Daml Ledger API. For example uses of the integration kit, see below.

9.2.2 Daml Integration Kit status and roadmap

The current status of the integration kit is ALPHA. We are working towards BETA, and General Availability (GA) will come quite a bit later. The roadmap below explains what we mean by these different statuses, and what's missing to progress.

ALPHA (current status) In the ALPHA status, the Daml integration kit is ready to be used by third-parties willing to accept the following caveats:

The architecture includes everything required to run Daml Applications using the Daml Ledger API. However, it misses support for testing Daml Applications in a uniform way against different Daml Ledgers.

Ledger API authorization, package upload, party on-boarding, ledger reset, and time manipulation are specific to each Daml Ledger, until the uniform *administrative Daml ledger access* API is introduced, which is different to the uniform *per-party Daml ledger access* that the Daml Ledger API provides. We will address this before reaching BETA status.

The architecture is likely to change due to learnings from integrators like you! Where possible we strive to make these changes backwards compatible. though this might not always be possible.

The documentation might be spotty in some places, and you might have to infer some of the documentation from the code.

Some of our code might be fresh off the press and might therefore have a higher rate of bugs.

That said: we highly value your feedback and input on where you find Daml software and this integration kit most useful. You can get into contact with us using the feedback form on this documentation page or by creating issues or pull-requests against the digital-asset/daml GitHub repository.

BETA For us, BETA status means that we have architectural stability and solid documentation in place. At this point, third-parties should have everything they need to integrate Daml with their ledger of choice completely on their own.

Before reaching BETA status, we expect to have:

hardened our test tooling
built tooling for benchmarking Daml ledgers
completed several integrations of Daml for different ledgers
implemented uniform *administrative Daml ledger access* to provide a portable way for testing Daml applications against different Daml ledgers

Related links

Tracking GitHub issue
GitHub milestone tracking work to reach BETA status

GA For us GA (General Availability) means that there are several production-ready Daml ledgers built using the Daml integration kit. We expect to reach GA in 2019.
Related links
Tracking GitHub issue

9.2.3 Implementing your own Daml Ledger

Each *X* ledger requires at least the implementation of a specific `daml-on-<X>-server`, which implements the Daml Ledger API. It might also require the implementation of a `<X>-daml-validator`, which provides the ability for nodes to validate Daml transactions.

For more about these parts of the architecture, read the *Architectural overview*.

9.2.3.1 Step-by-step guide

Prerequisite knowledge

Before you can decide on an appropriate architecture and implement your own server and validator, you need a significant amount of context about Daml. To acquire this context, you should:

1. Complete the *IOU Quickstart Tutorial*.
2. Get an in-depth understanding of the *Daml Ledger Model*.
3. Build a mental model of how the *Ledger API* is used to *build Daml Applications*.

Deciding on the architecture and writing the code

Once you have the necessary context, we recommend the steps to implement your own server and validator:

1. Clone our example Daml Ledger (which is backed by an in-memory key-value store) from the digital-asset/daml-on-x-example.

1. Read the example code jointly with the *Architectural overview*, *Resources we provide*, and the *Library infrastructure overview* below.

1. Combine all the knowledge gained to decide on the architecture for your Daml on *X* ledger.

1. Implement your architecture; and let the world know about it by creating a PR against the digital-asset/daml repository to add your ledger to the list of *Daml Ledgers built or in development*.

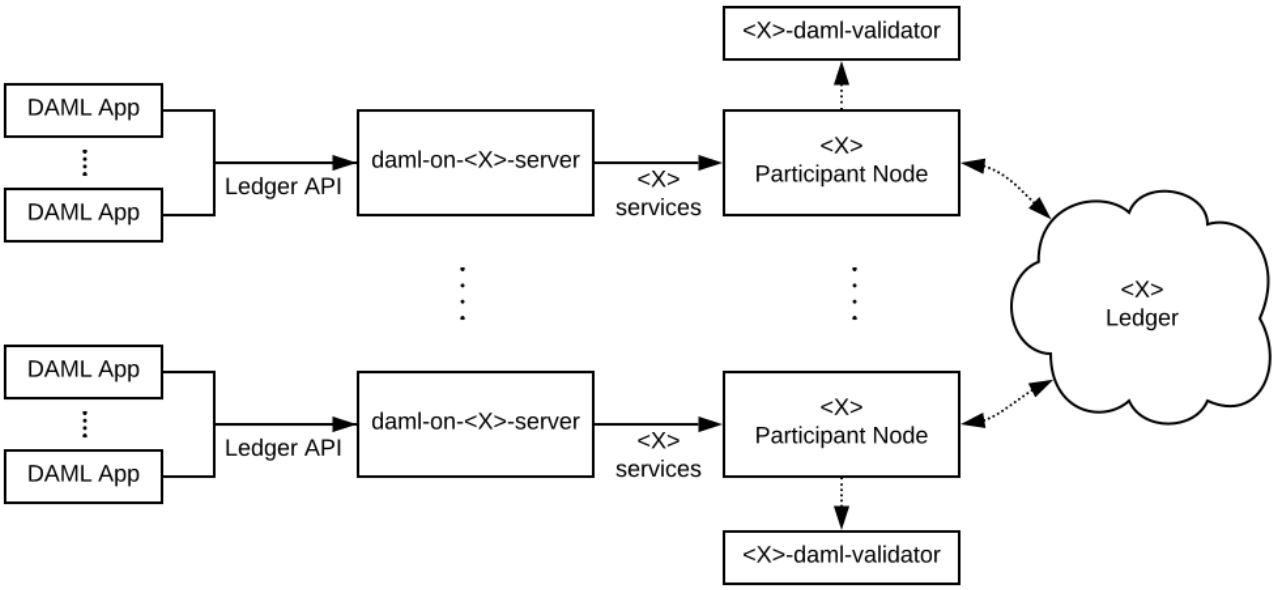
If you need help, then feel free to use the feedback form on this documentation page or GitHub issues on the digital-asset/daml repository to get into contact with us.

9.2.3.2 Architectural overview

This section explains the architecture of a Daml ledger backed by a specific ledger *X*.

The backing ledger can be a proper distributed ledger or also just a database. The goal of a Daml ledger implementation is to allow multiple Daml applications, which are potentially run by different entities, to execute multi-party workflows using the ledger *X*.

This is a likely architecture for a setup with a distributed ledger:



It assumes that the *X* ledger allows entities to participate in the evolution of the ledger via particular nodes. In the remainder of this documentation, we call these nodes *participant nodes*.

In the diagram:

The boxes labeled *daml-on-<X>-server* denote the Daml Ledger API servers, which implement the Daml Ledger API on top of the services provided by the *X* participant nodes.
The boxes labeled *<X>-daml-validator* denote *X*-specific Daml transaction validation services. In a distributed ledger they provide the ability for nodes to *validate Daml transactions* at the appropriate stage in the *X* ledger's transaction commit process.
Whether they are needed, by what nodes they are used, and whether they are run in-process or out-of-process depends on the *X* ledger's architecture. Above we depict a common case where the participant nodes jointly maintain the ledger's integrity and therefore need to validate Daml transactions.

Message flow

TODO (BETA):

explain to readers the life of a transaction at a high-level, so they have a mental framework in place when looking at the example code. (GitHub issue)

9.2.3.3 Resources we provide

Scala libraries for validating Daml transactions and serving the Ledger API given implementations of two specific interfaces. See the *Library infrastructure overview* for an overview of these libraries.
A complete example of a Daml Ledger backed by an in-memory key-value store, in the digital-asset/daml-on-x-example GitHub repository. It builds on our Scala libraries and demonstrates how they can be assembled to serve the Ledger API and validate Daml transactions.
For ledgers where data is shared between all participant nodes, we recommend using this example as a starting point for implementing your server and validator.
For ledgers with stronger privacy models, this example can serve as an inspiration. You will need to dive deeper into how transactions are represented and how to communicate them to implement *Daml's privacy model* at the ledger level instead of just at the Ledger API level.

Library infrastructure overview

To help you implement your server and validator, we provide the following four Scala libraries as part of Daml Connect. Changes to them are explained as part of the *Release Notes*.

As explained in *Deciding on the architecture and writing the code*, this section is best read jointly with the code in digital-asset/daml-on-x-example.

`participant-state.jar` (source code) Contains interfaces abstracting over the state of a participant node relevant for a Daml Ledger API server.
These are the interfaces whose implementation is specific to a particular *X* ledger. These interfaces are optimized for ease of implementation.

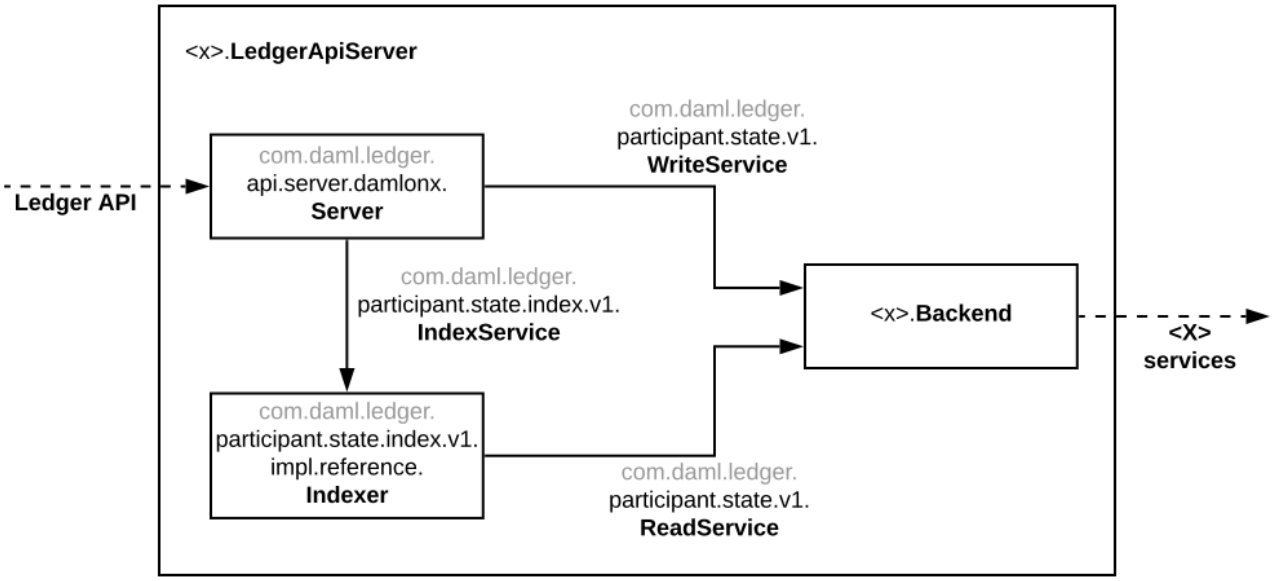
`participant-state-kvutils.jar` (source code) These utilities provide methods to succinctly implement interfaces from `participant-state.jar` on top of a key-value state storage.
See documentation in package.scala

`ledger-api-server.jar` (source code for API server, source code for indexer) Contains code that implements a Daml Ledger API server and the SQL-backed indexer given implementations of the interfaces in `participant-state.jar`.

`daml-engine.jar` (source code) Contains code for serializing and deserializing Daml transactions and for validating them.
An *<X>-daml-validator* is typically implemented by wrapping this code in the *X*-ledger's SDK for building transaction validators. `daml-engine.jar` also contains code for interpreting commands sent over the Ledger API. It is used by the *daml-on-<X>-server* to construct the transactions submitted to its participant node.

This diagram shows how the classes and interfaces provided by these libraries are typically combined to instantiate a Daml Ledger API server backed by an *X* ledger:

TODO: Update this diagram to mention ledger server classes above instead of deprecated daml-on-x-server



In the diagram above:

Boxes labeled with fully qualified class names denote class instances.
Solid arrows labeled with fully qualified interface names denote that an instance depends on another instance providing that interface.
Dashed arrows denote that a class instance provides or depends on particular services.

Boxes embedded in other boxes denote that the outer class instance creates the contained instances.

Explaining this diagram in detail (for brevity, we drop prefixes of their qualified names where unambiguous):

Ledger API is the collection of gRPC services that you would like your *daml-on-<X>-server* to provide.

<X> services are the services provided by which underly your ledger, which you aim to leverage to build your *daml-on-<X>-server*.

<x>.LedgerApiServer is the class whose main method or constructor creates the contained instances and wires them up to provide the Ledger API backed by the `<X> services`. You need to implement this for your Daml on *X* ledger.

WriteService (source code) is an interface abstracting over the mechanism to submit Daml transactions to the underlying *X* ledger via a participant node.

ReadService (source code) is an interface abstracting over the ability to subscribe to changes of the *X* ledger visible to a particular participant node. The changes are exposed as a stream that is resumable from any particular offset, which supports restarts of the consumer. We typically expect there to be a single consumer of the data provided on this interface. That consumer is responsible for assembling the streamed changes into a view onto the participant state suitable for querying.

<x>.Backend is a class implementing the `ReadService` and the `WriteService` on top of the `<X> services`. You need to implement this for your Daml on *X* ledger.

StandaloneIndexerServer (source code) is a standalone service that subscribe to ledger changes using `ReadService` and inserts the data into a SQL backend (index) for the purpose of serving the data over the Ledger API.

StandaloneIndexServer (source code) is a class containing all the code to implement the Ledger API on top of an ledger backend. It serves the data from a SQL database populated by the `StandaloneIndexerServer`.

9.2.4 Deploying a Daml Ledger

TODO (BETA):

explain recommended approach for Ledger API authorization (GitHub issue)
explain option of using a persistent SQL-backed participant state index (GitHub issue).
explain how testing of Daml applications (ledger reset, time manipulation, scripted package upload) can be supported by a uniform admin interface (GitHub issue).

9.2.4.1 Authorization

To implement authorization on your ledger, do the following modifications to your code:

Implement the `com.daml.ledger.api.auth.AuthService` (source code) interface. An AuthService receives all HTTP headers attached to a gRPC ledger API request and returns a set of `Claims` (source code), which describe the authorization of the request.
Instantiate a `com.daml.ledger.api.auth.interceptor.AuthorizationInterceptor` (source code), and pass it an instance of your AuthService implementation. This interceptor will be responsible for storing the decoded Claims in a place where ledger API services can access them.
When starting the `com.daml.platform.apiserver.LedgerApiServer` (source code), add the above AuthorizationInterceptor to the list of interceptors (see `interceptors` parameter of `LedgerApiServer.create`).

For reference, you can have a look at how authorization is implemented in the sandbox:

The `com.daml.ledger.api.auth.AuthServiceJWT` class (source code) reads a JWT token from HTTP headers.
The `com.daml.ledger.api.auth.AuthServiceJWTPayload` class (source code) defines the format of the token payload.
The token signature algorithm and the corresponding public key is specified as a sandbox command line parameter.

9.2.5 Testing a Daml Ledger

You can test your Daml ledger implementation using *Ledger API Test Tool*, which will assess correctness of implementation of the *Ledger API*. For example, it will show you if there are consistency or conformance problem with your implementation.

Assuming that your Ledger API endpoint is accessible at `localhost:6865`, you can use the tool in the following manner:

1. Download the Ledger API Test Tool from Maven and save it as `ledger-api-test-tool.jar` in your current directory.
2. Obtain the Daml archives required to run the tests:
   ```
   java -jar ledger-api-test-tool.jar --extract
   ```
3. Load all `.dar` files extracted in the current directory into your Ledger.
4. Run the tool against your ledger:
   ```
   java -jar ledger-api-test-tool.jar localhost:6865
   ```

See more in *Ledger API Test Tool*.

9.2.6 Benchmarking a Daml Ledger

TODO (BETA):

explain how to use the `ledger-api-bench` tool to evaluate the performance of your implementation of the Ledger API (GitHub issue).

9.3 Daml Triggers - Off-Ledger Automation in Daml

9.3.1 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.

9.3.1.1 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.

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) (ACS -> 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) ((Map *CommandId* [*Command*], ACS) -> 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`.

9.3.1.2 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 -> *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.

9.3.1.3 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 -> *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	TemplateTypeRep	
contractKey	AnyContractKey	
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 contractId *Command AnyContractId*

instance HasField contractKey *Command* AnyContractKey

instance HasField runTriggerUpdateA (*TriggerUpdateA* s a) ((Map *CommandId* [*Command*], ACS) -> State s a)

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 pendingContracts ACS (Map *CommandId* [*AnyContractId*])

instance HasField pendingContracts (TriggerAState s) (Map *CommandId* [*AnyContractId*])

instance HasField runTriggerUpdateA (*TriggerUpdateA* s a) ((Map *CommandId* [*Command*], ACS) -> State s a)

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	*Completion-Status*	

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 -> *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 -> *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 -> *ActiveContracts* -> *TriggerSetup* s)

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

Functions

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

fromAnyContractId : Template t => *AnyContractId* -> Optional (ContractId t)
Check if a `AnyContractId` corresponds to the given template or return `None` otherwise.

fromCreated : Template t => *Created* -> Optional (*EventId*, ContractId t, t)
Check if a `Created` event corresponds to the given template.

fromArchived : Template t => *Archived* -> Optional (*EventId*, ContractId t)
Check if an `Archived` event corresponds to the given template.

registeredTemplate : Template t => RegisteredTemplate

createCmd : Template t => t -> *Command*
Create a contract of the given template.

exerciseCmd : Choice t c r => ContractId t -> c -> *Command*
Exercise the given choice.

createAndExerciseCmd : (Template t, Choice t c r) => t -> c -> *Command*
Create a contract of the given template and immediately exercise the given choice on it.

exerciseByKeyCmd : (Choice t c r, TemplateKey t k) => k -> c -> *Command*

fromCreate : Template t => *Command* -> Optional t
Check if the command corresponds to a create command for the given template.

fromCreateAndExercise : (Template t, Choice t c r) => *Command* -> Optional (t, c)
Check if the command corresponds to a create and exercise command for the given template.

fromExercise : Choice t c r => *Command* -> Optional (ContractId t, c)
Check if the command corresponds to an exercise command for the given template.

fromExerciseByKey : (Choice t c r, TemplateKey t k) => *Command* -> Optional (k, c)
Check if the command corresponds to an exercise by key command for the given template.

execStateT : Functor m => StateT s m a -> s -> m s

zoom : Functor m => (t -> s) -> (t -> s -> t) -> StateT s m a -> StateT t m a

simulateRule : *TriggerRule* s a -> Time -> s -> (s, [*Commands*], a)
Run a rule without running it. May lose information from the rule; meant for testing purposes only.

submitCommands : *ActionTrigger* m => [*Command*] -> m *CommandId*

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-name=gsg-trigger create-daml-app
```

9.3.2 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.

9.3.3 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 .

9.3.4 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`:

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.

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.

9.3.5 Running a No-Op Trigger

To implement a no-op trigger, one could write the following in a separate `daml/ChatBot.daml` file:

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/create-daml-app-0.1.0.dar \
             --trigger-name NoOp:noOp \
             --ledger-host localhost \
             --ledger-port 6865 \
             --ledger-party "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.

9.3.6 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:

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`:

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.

9.3.7 AutoReply

Open up the trigger code again (`daml/ChatBot.daml`), and change it to:

Refresh `daml start` by pressing `r` (followed by `Enter` on Windows) in its terminal, then start the trigger with:

```
daml trigger --dar .daml/dist/create-daml-app-0.1.0.dar \
             --trigger-name ChatBot:autoReply \
             --ledger-host localhost \
             --ledger-port 6865 \
             --ledger-party "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.

9.3.8 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.

9.3.9 Authentication

When using Daml triggers against a Ledger with authentication, 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 trigers for more than one party at a time, you may be interested in the /tools/trigger-service/index.

9.3.10 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.

9.4 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:

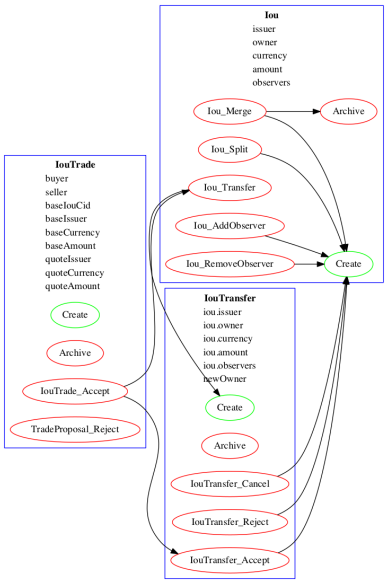
1. Install Graphviz.
2. Generate a DAR from your project by running `daml build`.
3. Generate a dot file from that DAR by running `daml damlc visual <path_to_project>/dist/<project_name.dar> --dot <project_name>.dot`
4. Generate the visual graph with Graphviz by running `dot -Tpng <project_name>.dot > <project_name>.png`

9.4.1 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:



9.4.2 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.

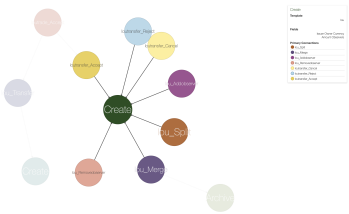
Note: You will need to install the Graphviz/dot packages as mentioned above.

9.4.3 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:



9.5 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.

9.5.1 Interoperability examples

9.5.1.1 Topology

Participant Nodes connect to Daml ledgers and parties access projections of these ledgers via the Ledger API. The following picture shows such a setup.

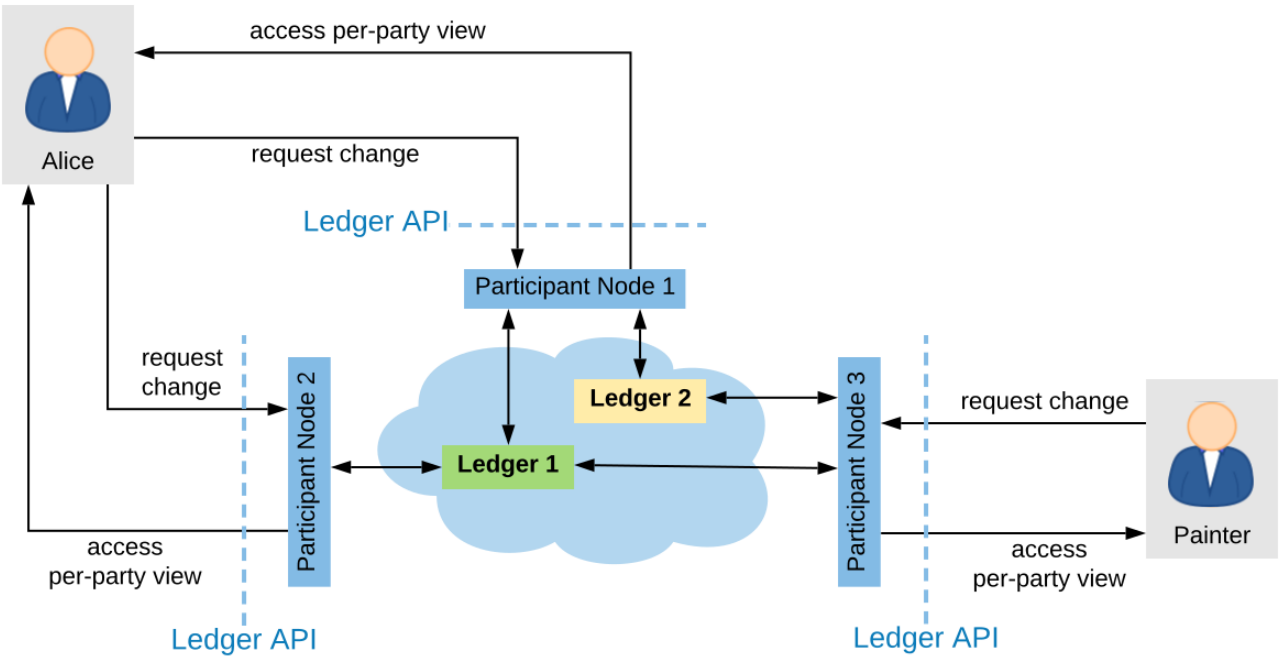


Fig. 1: Example topology with three interoperable ledgers

The components in this diagram are the following:

There is a set of interoperable **Daml ledgers**: Ledger 1 (green) and Ledger 2 (yellow).
Each **Participant Node** is connected to a subset of the Daml ledgers.
- Participant Nodes 1 and 3 are connected to Ledger 1 and 2.
- Participant Node 2 is connected to Ledger 1 only.

Participant Nodes host parties on a subset of the Daml ledgers they are connected to. A Participant Node provides a party access to the Daml ledgers that it hosts the party on.
- Participant Node 1 hosts Alice on Ledger 1 and 2.
- Participant Node 2 hosts Alice on Ledger 1.
- Participant Node 3 hosts the painter on Ledger 1 and 2.

9.5.1.2 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.

9.5.1.3 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.

9.5.1.4 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.

9.5.2 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.

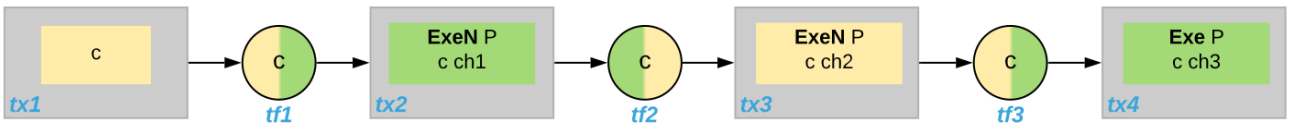


Fig. 2: Multi-Ledger causality graph with transfer actions

Note: As for ordinary causality graphs, the diagrams for multi-ledger causality graphs omit transi-

tive 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.

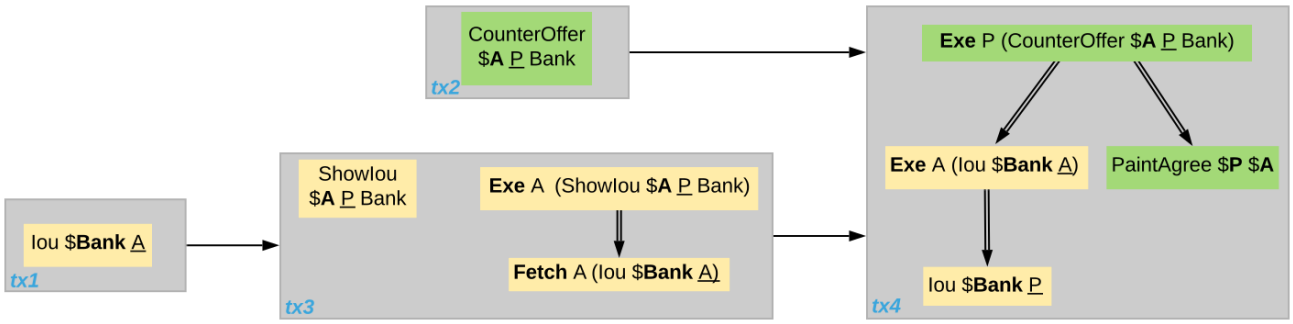


Fig. 3: Multi-Ledger causality graph for the split paint counteroffer workflow on two Daml ledgers

9.5.2.1 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**

9.5.2.2 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.

9.5.2.3 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.

9.5.3 Ledger-aware projection

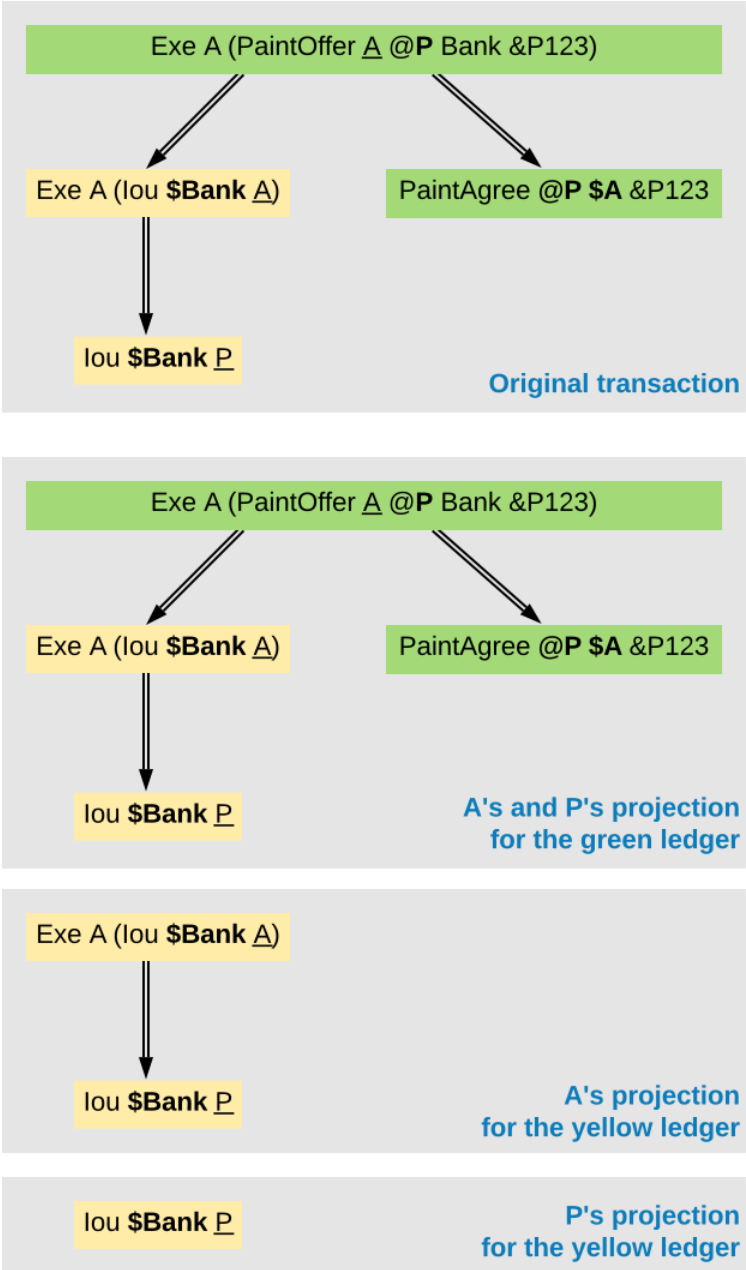
A Participant Node maintains a local ledger for each party it hosts and the Transaction Service outputs a topological sort of this local ledger. When the Participant Node hosts the party on several ledgers, this local ledger is an multi-ledger causality graph. This section defines the ledger-aware projection 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.

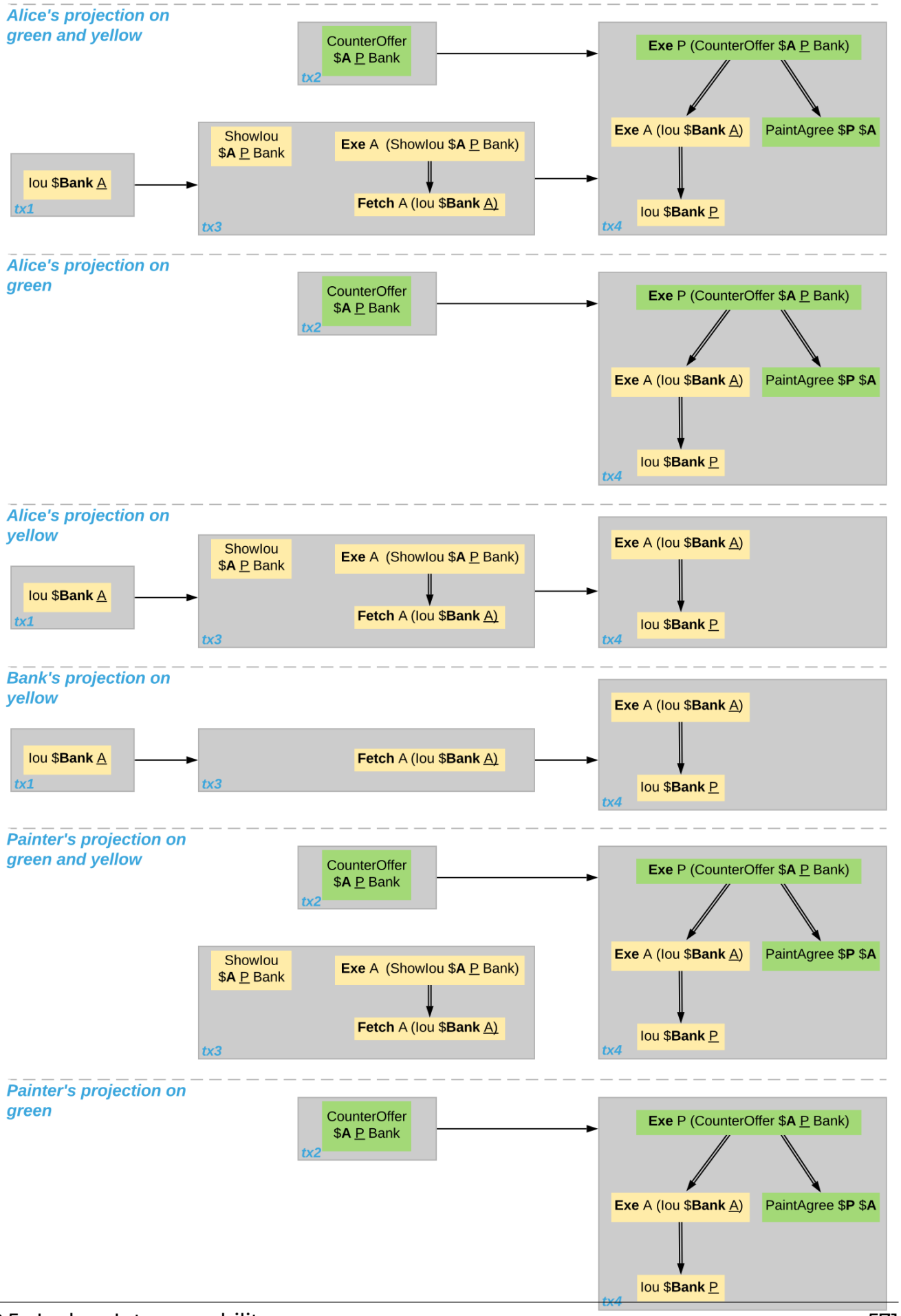
The notions of *X*-minimality and *X*-reduction extend to a party *P* on a set *Y* of ledgers accordingly.

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.

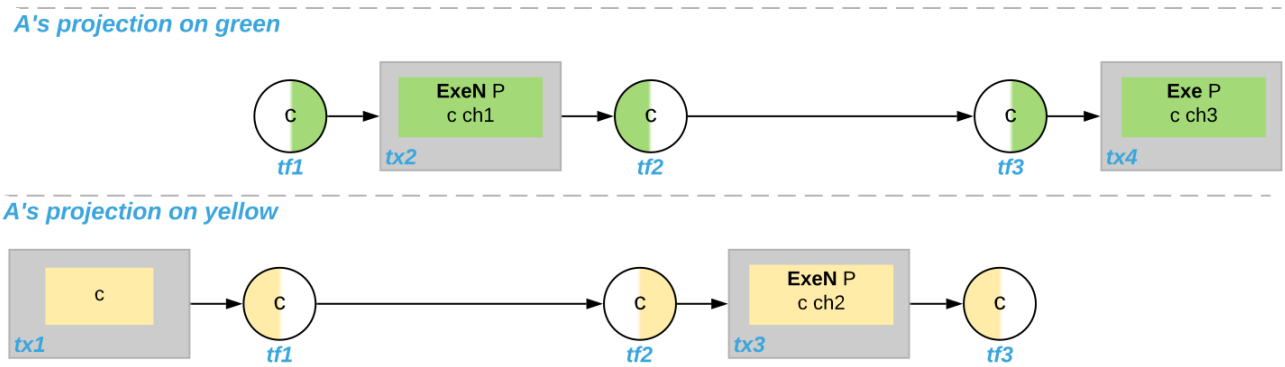
For example, the *multi-ledger causality graph for the split paint counteroffer workflow* is projected as follows.


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)
Exe A (Iou $Bank A)

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.



9.5.4 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.

Chapter 10

Daml Ecosystem

10.1 Daml Ecosystem Overview

10.1.1 Status Definitions

Throughout the documentation, we use labels to mark features of APIs not yet deemed stable. This page gives meaning to those labels.

10.1.1.1 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.

10.1.1.2 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.

10.1.1.3 Comparison of Statuses

The table below gives a concise overview of the labels used for Daml features and components.

Table 1: 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.

10.1.2 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*.

10.1.2.1 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

10.1.2.2 Integration Components

Component/Feature	Status	Deprecated on
Integration Kit Components	Labs	
CLI and test names of *Ledger API Test Tool*	Beta	

10.1.2.3 Runtime components

Component / Feature	Status	Deprecated on
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	Beta	
Development CLI to start individual triggers in dev environment (`daml trigger`)	Beta	
/tools/trigger-service/index (`daml trigger-service`)	Alpha	
Extractor		
Extractor (`daml extractor`)	Labs	

10.1.2.4 Libraries

Component / Feature	Status	Deprecated on
Scala Ledger API Bindings		
`daml codegen scala` *CLI and generated code*	Stable, Deprecated	2020-10-14
`bindings-scala_2.12` *library and its public API*	Stable, Deprecated	2020-10-14
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* *excluding* the reactive components in package `com.daml.ledger.rxjava.components`.	Stable	
Java Reactive Components in the `com.daml.ledger.rxjava.components` package of `bindings-rxjava`.	Stable, Deprecated	2020-10-14
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	
Node.js Ledger API Bindings		
`@digital-asset/bindings-js` *Node.js library*	Stable, Deprecated	2020-10-14
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	

10.1.2.5 Developer Tools

Component / Feature	Status	Deprecated on
SDK		
Windows SDK (installer)	Stable	
Mac SDK	Stable	

Continued on next page

Table 2 – continued from previous page

Component / Feature	Status	Deprecated on
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`, `daml trigger-service`, and `daml extractor`.	See *Runtime components*.	
Daml Projects		
`daml.yaml` project specification	Stable	
Assistant commands `new`, `create-daml-app`, and `init`. Note that the templates created by `daml new` and `create-daml-app` are considered example code, and are not covered by *semantic versioning*.	Stable	
Daml Studio		
VSCode Extension	Stable	
`daml studio` assistant command	Stable	
Code Generation		
`daml codegen` assistant commands	See *Libraries*.	
Sandbox Development Ledger		
`daml sandbox` assistant command and documented CLI under `daml sandbox --help`.	Stable	
Daml Sandbox in Memory (ie without the `--sql-backend-jdbcurl` flag)	Stable	
Daml Sandbox on Postgres (ie with the `--sql-backend-jdbcurl` flag)	Stable, Deprecated	2020-12-16
Daml Sandbox Classic and associated CLIs `daml sandbox-classic`, `daml start --sandbox-classic`	Stable, Deprecated	2020-04-09
Daml Profiler `daml sandbox --profile-dir`	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	

Continued on next page

Table 2 – continued from previous page

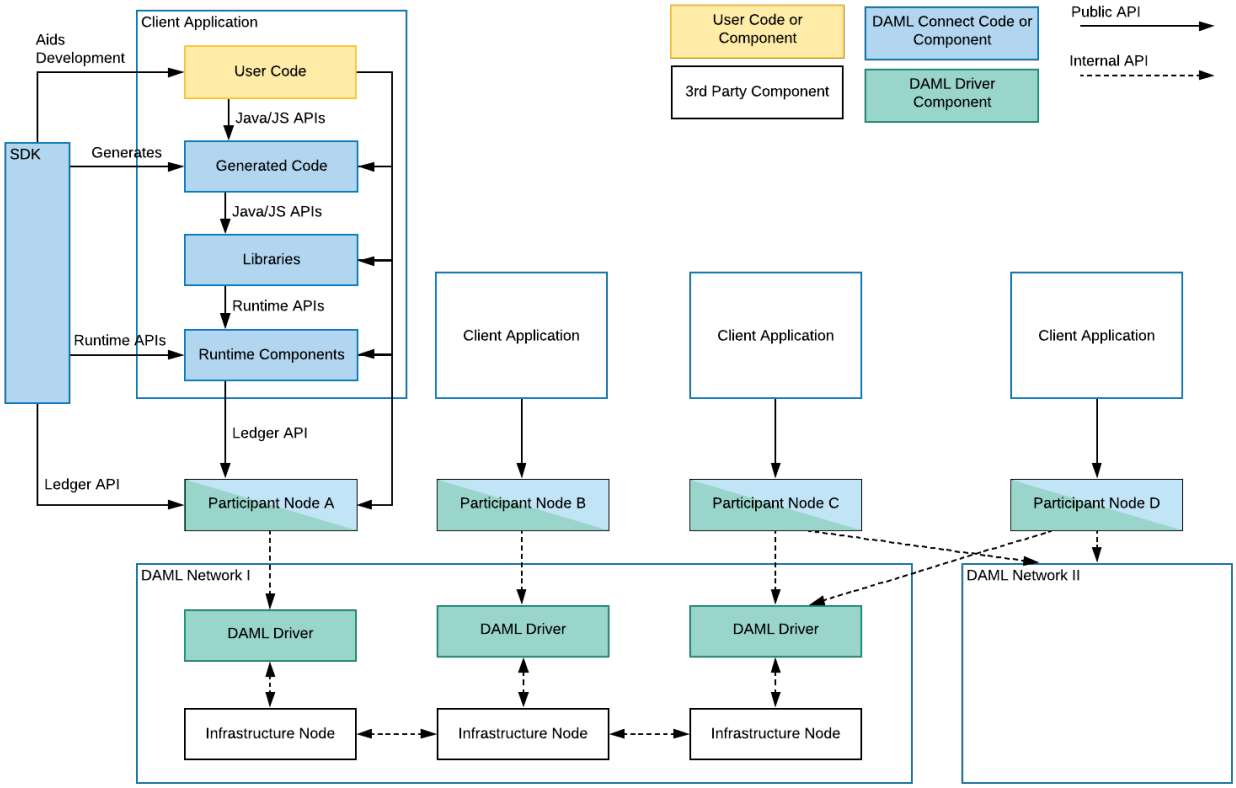
Component / Feature	Status	Deprecated on
DAR File inspection (`daml damlc inspect-dar`). The exact output is only covered by *semantic versioning* when used with the `--json` flag.	Stable	
DAR File validation (`daml damlc validate-dar`)	Stable	
Daml Linter (`daml damlc lint`)	Stable	
Daml REPL (`daml 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	
Scenarios and Script		
Scenario Daml API	Stable	
Script Daml API	Stable	
Daml Scenario IDE integration	Stable	
Daml Script IDE integration	Stable	
Daml Script Library	See *Libraries*	
`daml test` in-memory Script and Scenario test CLI	Stable	
`daml test-script` Sandbox-based Script Testing	Stable	
`daml script` *CLI to run Scripts against live ledgers.*	Stable	
Navigator		
Daml Navigator Development UI (`daml navigator server`)	Stable	
Navigator Config File Creation (`daml navigator create-config`)	Stable	
Navigator graphQL Schema (`daml navigator dump-graphql-schema`)	Labs	
Daml REPL Interactive Shell		
`daml repl` *CLI*	Stable	
Daml and meta-APIs of the REPL	Stable	
Ledger Administration CLI		
`daml ledger` *CLI and all subcommands*.	Stable	

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.

10.1.3 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 Connect** consists of everything developers and users need to connect to a Daml Network: the tools to build, deploy, integrate, and maintain a Daml Application.

10.1.3.1 Daml Networks

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 command line interface of Daml for Postgres. It's a public interface, but specific to the Postgres driver.

Integration Components

Daml drivers and Participant Nodes share a lot of components between underlying DLTs or Databases. These shared components are called the Integration Components, or sometimes the *Daml Integration Kit*.

10.1.3.2 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.

10.1.3.3 Ledger API

The Ledger API is the primary interface that offers forward and backward compatibility between Daml Networks and Applications (including Daml Connect 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.

10.1.3.4 Daml Connect

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.

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*.

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 Connect version and source of the generated code, like a Daml package. In this case, the version of the Daml Connect SDK used covers changes to the public API of the generated code.

Developer Tools / SDK

The Daml Connect 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*.

10.2 Releases and Versioning

10.2.1 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*.

10.2.2 Cadence

Regular snapshot releases are made every Wednesday, with additional snapshots released as needed. These releases contain Daml Connect and Integration 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, and additional releases may be made, or releases may be delayed for 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.

10.2.3 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.

10.2.4 Release Notes

Release notes for each release are published on the Release Notes section of the Daml Driven blog.

10.2.5 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.

10.2.6 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**.

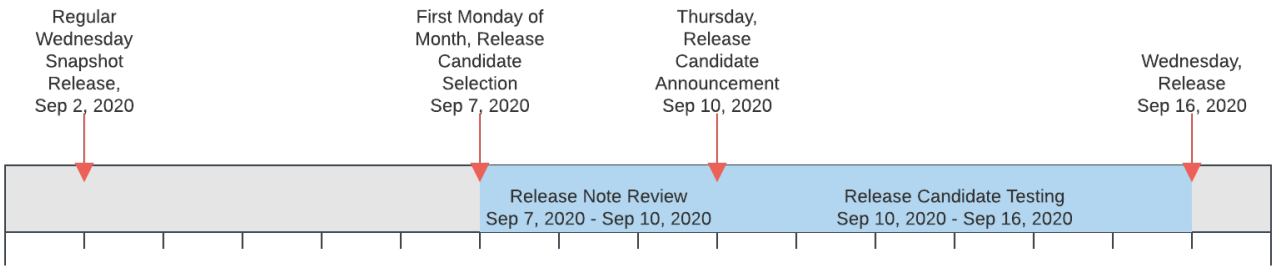


Fig. 1: The release process timeline illustrated by example of September 2020.

10.3 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 Connect 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.

10.3.1 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.

10.3.1.1 List of Ledger API Versions supported by Daml Connect

The below lists with which Daml Connect version a new Ledger API version was introduced.

Ledger API Version	Daml Connect Version
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 Connect / SDK version

10.3.1.2 Summary of Ledger API Changes

Ledger API Version	Changes
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 Connect (/SDK) release notes of same version number.

10.3.2 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.

10.3.3 SDK, Runtime Component, and Library Compatibility: Daml Connect Upgradeability

As long as a major Ledger API version is supported (see *Ledger API Support Duration*), there will be supported version of Daml Connect 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 Connect 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.

10.3.4 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 Connect 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 Connect U.V.W against Ledger API X.Y.Z can be maintained using any Daml Connect 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.

10.4 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.

10.4.1 Support expectations

For community users (ie on our Forum and Stack Overflow):

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.