DAML SDK Documentation

Digital Asset

Version : 2019-11-07

Copyright 2019 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 Next steps . . . 1
  - 1.1.4 Alternative: manual download . . . 1
- 1.2 Quickstart guide . . . 3
  - 1.2.1 Download the quickstart application . . . 3
  - 1.2.2 Overview of what an IOU is . . . 4
  - 1.2.3 Run the application using prototyping tools . . . 5
  - 1.2.4 Try out the application . . . 6
  - 1.2.5 Get started with DAML . . . 10
  - 1.2.6 Integrate with the ledger . . . 15
  - 1.2.7 Next steps . . . 18

2 Writing DAML **19**
- 2.1 An introduction to DAML . . . 19
  - 2.1.1 1 Basic contracts . . . 19
  - 2.1.2 2 Testing templates using scenarios . . . 21
  - 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 . . . 47
  - 2.1.6 6 Parties and authority . . . 56
  - 2.1.7 7 Composing choices . . . 65
- 2.2 Language reference docs . . . 73
  - 2.2.1 Overview: template structure . . . 73
  - 2.2.2 Reference: templates . . . 76
  - 2.2.3 Reference: choices . . . 79
  - 2.2.4 Reference: updates . . . 81
  - 2.2.5 Reference: data types . . . 85
  - 2.2.6 Reference: built-in functions . . . 91
  - 2.2.7 Reference: expressions . . . 93
  - 2.2.8 Reference: functions . . . 95
  - 2.2.9 Reference: scenarios . . . 98
  - 2.2.10 Reference: DAML file structure . . . 99
  - 2.2.11 Reference: DAML packages . . . 100
  - 2.2.12 Contract keys . . . 101
  - 2.2.13 Generic Templates . . . 103

2.3 DAML Studio . . . 105
2.3.1 Installing . . . 105
2.3.2 Creating your first DAML file . . . 105
2.3.3 Supported features . . . 107
2.3.4 Common scenario errors . . . 109
2.4 Testing using scenarios . . . 113
2.4.1 Scenario syntax . . . 113
2.4.2 Running scenarios in DAML Studio . . . 114
2.4.3 Examples . . . 114
2.5 Troubleshooting . . . 116
2.5.1 Error: <X> is not authorized to commit an update . . . 116
2.5.2 Error Argument is not of serializable type . . . 116
2.5.3 Modelling questions . . . 117
2.5.4 Testing questions . . . 119
2.6 Writing good DAML . . . 119
2.6.1 Good design patterns . . . 119
2.6.2 Anti-patterns . . . 137
2.6.3 What functionality belongs in DAML models versus application code? . . . 141

3 Building applications **143**
3.1 Writing applications using the Ledger API . . . 143
3.1.1 The Ledger API services . . . 143
3.1.2 How DAML types are translated to DAML-LF . . . 147
3.1.3 Resources available to you . . . 152
3.1.4 What's in the Ledger API . . . 152
3.1.5 DAML-LF . . . 153
3.2 Java bindings . . . 154
3.2.1 Generate Java code from DAML . . . 154
3.2.2 Example project . . . 165
3.2.3 Overview . . . 167
3.2.4 Reference documentation . . . 169
3.2.5 Getting started . . . 169
3.2.6 Example project . . . 170
3.3 Scala bindings . . . 170
3.3.1 Introduction . . . 170
3.3.2 Getting started . . . 171
3.3.3 Generating Scala code . . . 171
3.3.4 Example code . . . 172
3.4 Node.js bindings . . . 173
3.5 The Ledger API using gRPC . . . 174
3.5.1 Ledger API Reference . . . 174
3.5.2 How DAML types are translated to protobuf . . . 204
3.5.3 Getting started . . . 211
3.5.4 Protobuf reference documentation . . . 211
3.5.5 Example project . . . 211
3.5.6 DAML types and protobuf . . . 212
3.5.7 Error handling . . . 212
3.6 Creating your own bindings . . . 212
3.6.1 Introduction . . . 212
3.6.2 Building Ledger Commands . . . 213
3.6.3 Summary . . . 215

3.6.4 Links . . . 215
3.7 Application architecture guide . . . 215
3.7.1 Categories of application . . . 216
3.7.2 Structuring an application . . . 216
3.7.3 Application libraries . . . 219
3.7.4 Architecture guidance . . . 219
3.7.5 Commonly used types . . . 221
3.7.6 Test the business logic with a ledger . . . 222
3.7.7 Share the ledger . . . 222
3.7.8 Reset if you need to . . . 222

4 SDK tools **224**
4.1 DAML Assistant (`daml`) . . . 224
4.1.1 Moving to the `daml` assistant . . . 224
4.1.2 Full help for commands . . . 224
4.1.3 Configuration files . . . 225
4.1.4 Building DAML projects . . . 226
4.1.5 Managing SDK releases . . . 227
4.2 DAML Sandbox . . . 227
4.2.1 Running with persistence . . . 228
4.2.2 Command-line reference . . . 228
4.3 Navigator . . . 229
4.3.1 Navigator functionality . . . 229
4.3.2 Installing and starting Navigator . . . 229
4.3.3 Choosing a party / changing the party . . . 229
4.3.4 Logging out . . . 230
4.3.5 Viewing templates or contracts . . . 230
4.3.6 Using Navigator . . . 234
4.3.7 Authenticating Navigator . . . 237
4.3.8 Advanced usage . . . 237

5 Background concepts **241**
5.1 Glossary of concepts . . . 241
5.1.1 DAML . . . 241
5.1.2 SDK tools . . . 245
5.1.3 Building applications . . . 246
5.1.4 General concepts . . . 248
5.2 DA Ledger Model . . . 248
5.2.1 Structure . . . 249
5.2.2 Integrity . . . 256
5.2.3 Privacy . . . 268
5.2.4 DAML: Defining Contract Models Compactly . . . 276

6 Deploying **278**
6.1 Deploying to DAML ledgers . . . 278
6.1.1 How to deploy . . . 278
6.1.2 Available DAML Products . . . 279
6.1.3 Open Source Integrations . . . 279
6.1.4 DAML ledgers in development . . . 279

7 Examples **280**
7.1 DAML examples . . . 280

8 Experimental features 281
8.1 WARNING . . . 281
8.1.1 Navigator Console . . . 281
8.1.2 Extractor . . . 292
8.2 DAML Integration Kit - ALPHA . . . 302
8.2.1 Ledger API Test Tool . . . 302
8.2.2 DAML Integration Kit status and roadmap . . . 305
8.2.3 Implementing your own DAML Ledger . . . 306
8.2.4 Deploying a DAML Ledger . . . 309
8.2.5 Testing a DAML Ledger . . . 309
8.2.6 Benchmarking a DAML Ledger . . . 310
8.3 HTTP JSON API Service . . . 310
8.3.1 DAML-LF JSON Encoding . . . 310
8.3.2 /contracts/search query language . . . 317
8.3.3 How to start . . . 318
8.3.4 Example session . . . 319
8.4 DAML Triggers - Off-Ledger Automation in DAML . . . 323
8.4.1 DAML Trigger Library . . . 323
8.4.2 Usage . . . 331
8.4.3 When not to use DAML triggers . . . 335

9 Support and updates 336
9.1 Support . . . 336
9.1.1 Support expectations . . . 336
9.2 Release notes . . . 337
9.2.1 0.13.34 - 2019-11-07 . . . 337
9.2.2 0.13.33 - 2019-11-06 . . . 337
9.2.3 0.13.32 - 2019-10-29 . . . 338
9.2.4 0.13.31 - 2019-10-18 . . . 338
9.2.5 0.13.30 - 2019-10-15 . . . 339
9.2.6 0.13.29 - 2019-10-04 . . . 341
9.2.7 0.13.28 - 2019-10-04 . . . 341
9.2.8 0.13.27 - 2019-09-25 . . . 342
9.2.9 0.13.26 - 2019-09-24 . . . 342
9.2.10 0.13.25 - 2019-09-18 . . . 343
9.2.11 0.13.24 - 2019-09-16 . . . 343
9.2.12 0.13.23 - 2019-09-11 . . . 343
9.2.13 0.13.22 - 2019-09-04 . . . 344
9.2.14 0.13.21 - 2019-08-29 . . . 345
9.2.15 0.13.20 - 2019-08-22 . . . 345
9.2.16 0.13.19 - 2019-08-14 . . . 346
9.2.17 0.13.18 - 2019-08-07 . . . 347
9.2.18 0.13.17 - 2019-08-07 . . . 347
9.2.19 0.13.16 - 2019-08-01 . . . 347
9.2.20 0.13.15 - 2019-07-25 . . . 348
9.2.21 0.13.14 - 2019-07-22 . . . 348
9.2.22 0.13.13 - 2019-07-16 . . . 348
9.2.23 0.13.12 - 2019-07-09 . . . 350
9.2.24 0.13.11 - 2019-07-08 . . . 350
9.2.25 0.13.10 - 2019-06-28 . . . 351
9.2.26 0.13.9 - 2019-06-28 . . . 352

9.2.27 0.13.8 - 2019-06-27 . . . 352
9.2.28 0.13.7 - 2019-06-26 . . . 352
9.2.29 0.13.6 - 2019-06-25 . . . 353
9.2.30 0.13.5 - 2019-06-19 . . . 354
9.2.31 0.13.4 - 2019-06-19 . . . 354
9.2.32 0.13.3 - 2019-06-18 . . . 354
9.2.33 0.13.2 - 2019-06-18 . . . 355
9.2.34 0.13.1 - 2019-06-17 . . . 355
9.2.35 0.13.0 - 2019-06-17 . . . 355
9.2.36 0.12.25 — 2019-06-13 . . . 356
9.2.37 0.12.24 - 2019-06-06 . . . 356
9.2.38 0.12.23 - 2019-06-05 . . . 356
9.2.39 0.12.22 - 2019-05-29 . . . 357
9.2.40 0.12.21 - 2019-05-28 . . . 358
9.2.41 0.12.20 - 2019-05-23 . . . 358
9.2.42 0.12.19 - 2019-05-22 . . . 359
9.2.43 0.12.18 - 2019-05-20 . . . 359
9.2.44 0.12.17 - 2019-05-10 . . . 363
9.2.45 0.12.16 - 2019-05-07 . . . 363
9.2.46 0.12.15 - 2019-05-06 . . . 363
9.2.47 0.12.14 - 2019-05-03 . . . 364
9.2.48 0.12.13 - 2019-05-02 . . . 364
9.2.49 0.12.12 - 2019-04-30 . . . 364
9.2.50 0.12.11 - 2019-04-26 . . . 364
9.2.51 0.12.10 — 2019-04-25 . . . 365
9.2.52 0.12.9 — 2019-04-23 . . . 365
9.2.53 0.12.7 — 2019-04-17 . . . 365
9.2.54 0.12.6 — 2019-04-16 . . . 365
9.2.55 0.12.5 — 2019-04-15 . . . 365
9.2.56 0.12.4 — 2019-04-15 . . . 365
9.2.57 0.12.3 — 2019-04-12 . . . 365
9.2.58 0.12.2 — 2019-04-12 . . . 365
9.2.59 0.12.1 — 2019-04-04 . . . 366
9.2.60 0.12.0 — 2019-04-04 . . . 366
9.2.61 0.11.32 . . . 366
9.2.62 0.11.3 - 2019-02-07 . . . 367
9.2.63 0.11.2 - 2019-01-31 . . . 367
9.2.64 0.11.1 - 2019-01-24 . . . 367
9.2.65 0.11.0 - 2019-01-17 . . . 367
9.3 DAML roadmap (as of September 2019) . . . 371

Chapter 1

Getting started

1.1 Installing the SDK

1.1.1 1. Install the dependencies

The SDK currently runs on Windows, MacOS or Linux.

You need to install:

1. Visual Studio Code.
2. JDK 8 or greater.
   You can get the JDK from Zulu 8 JDK or Oracle 8 JDK (requires you to accept Oracle's license).

1.1.2 2. Install the SDK

1.1.2.1 Mac and Linux

To install the SDK on Mac or Linux:

1. Run:

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

2. If prompted, add `~/.daml/bin` to your PATH.
   If you don't know how to do this, try following these instructions for MacOS or these instructions for Windows.

1.1.2.2 Windows

We support running the SDK on Windows 10. To install the SDK on Windows, download and run the installer from github.com/digital-asset/daml/releases/latest.

1.1.3 Next steps

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

1.1.4 Alternative: manual download

The cURL command above will automatically download and run the DAML installation script from GitHub (using TLS). If you require a higher level of security, you can instead install the SDK by manu-

ally 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 0.13.27, you would download the files `daml-sdk-0.13.27-macos.tar.gz` and `daml-sdk-0.13.27-macos.tar.gz.asc`. Note that for Windows you can choose between the tarball, 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 pgp.key-server.io --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 2021-05-15. If any of those details are different, something is wrong. In that case please contact Digital Asset immediately.
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 v0.13.27 on macOS, you should have both files in the current directory and run:

   ```
   gpg --verify daml-sdk-0.13.27-macos.tar.gz.asc
   ```

   and that should give you a result that looks like:

   ```
   gpg: assuming signed data in 'daml-sdk-0.13.27-macos.tar.gz'
   gpg: Signature made Wed Sep 25 11:57:28 2019 BST
   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-sdl-0.13.27-macos.tar.gz
cd sdk-0.13.27
./install.sh
```

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

1.2 Quickstart guide

In this guide, you will learn about the SDK 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 DAML SDK. See *Installing the SDK*.

On this page:

Download the quickstart application
- *Folder structure*

Overview of what an IOU is
Run the application using prototyping tools
Try out the application
Get started with DAML
- *Develop with DAML Studio*
- *Test using scenarios*

Integrate with the ledger
Next steps

1.2.1 Download the quickstart application

You can get the quickstart application using the DAML assistant (`daml`):

1. Run `daml new quickstart quickstart-java`
   This creates the `quickstart-java` application into a new folder called `quickstart`.
2. Run `cd quickstart` to change into the new directory.

1.2.1.1 Folder structure

The project contains the following files:

```
.
├── daml
│   ├── Iou.daml
│   ├── IouTrade.daml
│   ├── Main.daml
│   └── Tests
```

(continues on next page)

(continued from previous page)

```
│        ├── 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.

1.2.2 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.

1.2.3 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 --scenario Main:setup .daml/dist/quickstart-0.0.1.dar`
   The output should look like this:

```
DAML LF Engine supports LF versions: 0, 1.0, 1.1, 1.2, 1.3; Transaction
↪versions: 1, 2, 3, 4, 5; Value versions: 1, 2, 3, 4
Starting plainText server
listening on localhost:6865
  ____              ____
  / __/__ ____  ___/ / /  ___ __ __
_\ \/  _ `/  _ \/  _  / _ \/ _ \\ \ /
/___/\_,_/_//_/\_,_/_.__/\___/_\_\

Initialized sandbox version 100.13.10 with ledger-id = sandbox-
↪5e12e502-817e-41f9-ad40-1c57b8845f9d, port = 6865, dar file =
↪DamlPackageContainer(List(target/daml/iou.dar),false), time mode =
↪Static, ledger = in-memory, daml-engine = {}
```

   The sandbox is now running, and you can access its *ledger API* on port `6865`.

   Note: The parameter `--scenario Main:setup` loaded the sandbox ledger with some initial data. Only the sandbox has this prototyping feature - it's not available on the full ledger server. More on *scenarios* later.

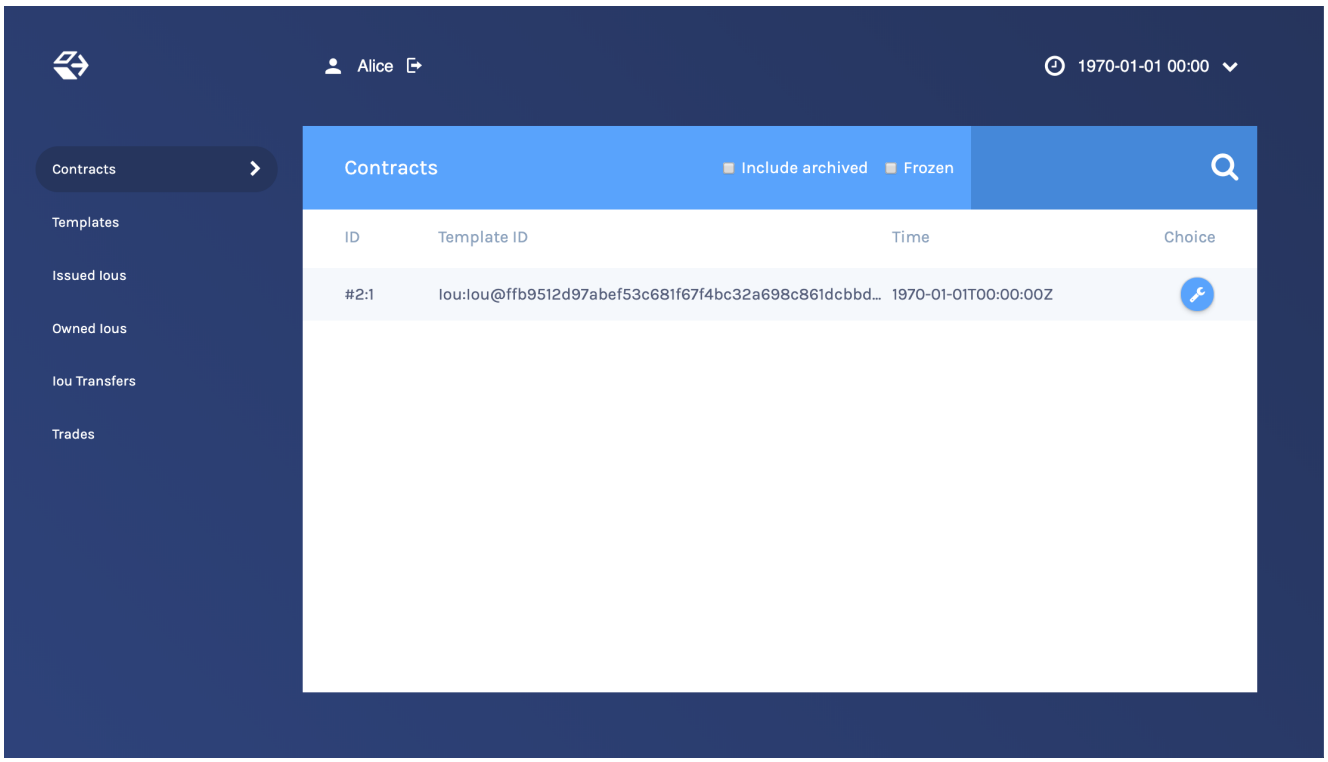
3. Open a new terminal window and navigate to your project directory, `quickstart`.
4. Start the *Navigator*, a browser-based leger front-end, by running `daml navigator server`
   The Navigator automatically connects the sandbox. You can access it on port `4000`.

1.2.4 Try out the application

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

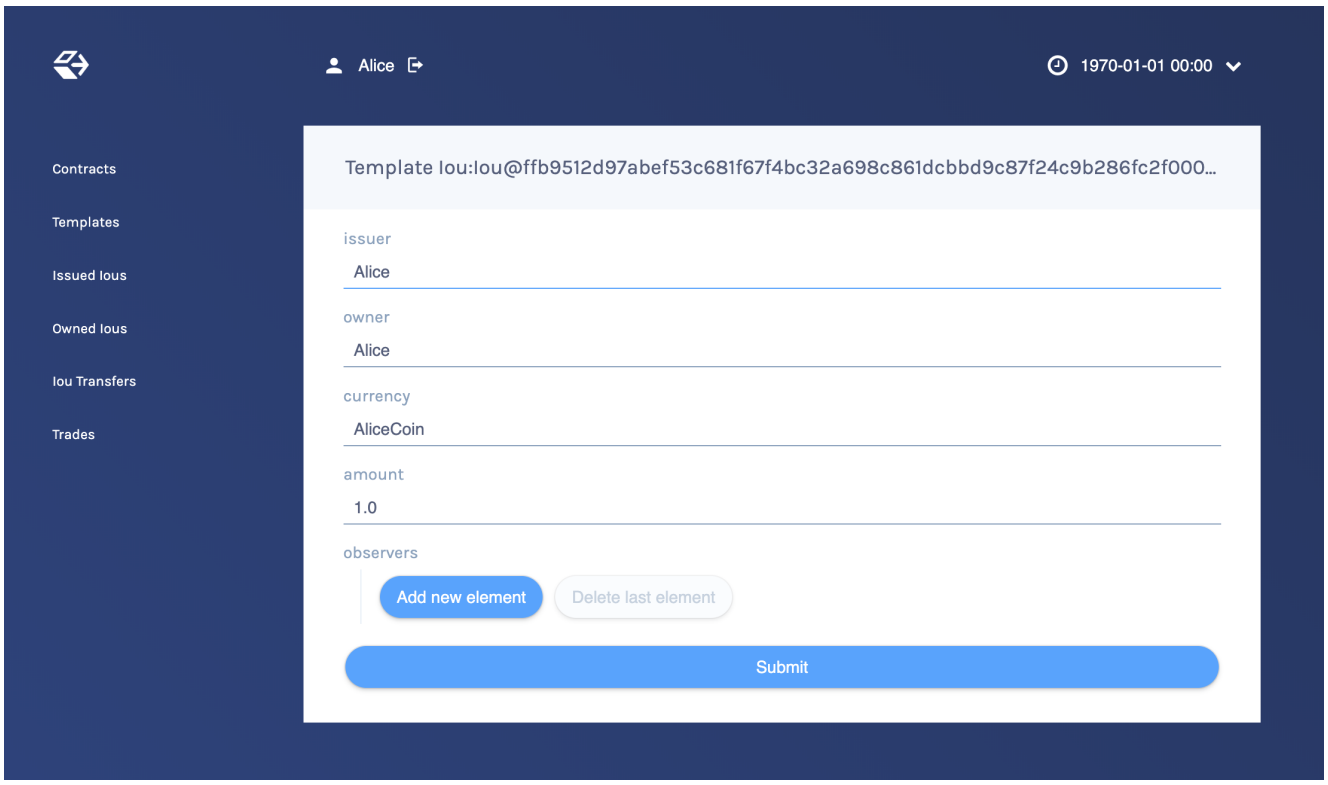
1. Go to http://localhost:4000/. This is the Navigator, which you launched *earlier*.
2. On the login screen, select **Alice** from the dropdown. This logs you in as Alice.
   (The list of available parties is specified in the `ui-backend.conf` file.)
   This takes you to the contracts view:
   This 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, with Id `#2:1`, created from a *template* called `Iou:Iou@ffb...`.
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 *contract instances* that you can see for each template.

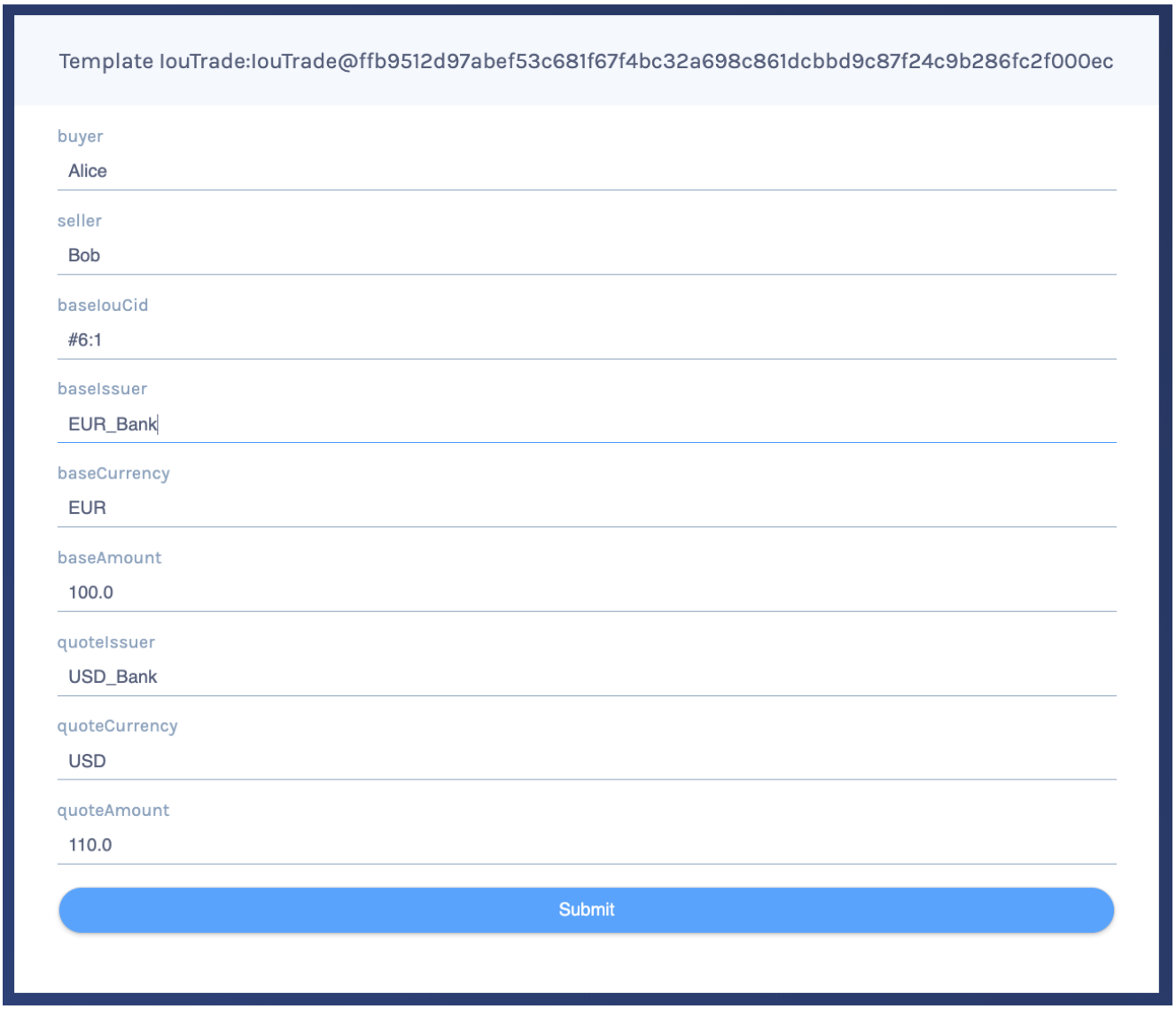
5. Try creating a contract from a template. Issue an Iou to yourself by clicking on the `Iou:Iou` row, filling it out as shown below and clicking **Submit**.
6. On the left-hand side, click **Issued Ious** to go to that page. You can see the Iou you just issued yourself.
7. Now, try transferring this 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 `#2:1` that you already saw on the *Contracts* page. It is an Iou for 100, issued by *EUR_Bank*.
9. Go to the **Iou Transfers** page. It shows the transfer of your recently issued Iou to Bob, but Bob has not accepted the transfer, so it is not settled.
   This is an important part of DAML: nobody can be forced into owning an *Iou*, or indeed agreeing to any other contract. They must explicitly consent.
   You could cancel the transfer by using the *IouTransfer_Cancel* choice within it, but for this walk-through, leave it alone for the time being.
10. Try asking *Bob* to exchange your 100 for $110. To do so, you first have to show your Iou to *Bob* so that he can verify the settlement transaction, should he accept the proposal.
    Go back to *Owned Ious*, open the Iou for 100 and click on the button *Iou_AddObserver*. Submit *Bob* as the *newObserver*.
    Contracts in DAML are immutable, meaning they can not 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 *#6:1*.
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* `#3:1`.
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 `#3:1` 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 three contracts:
    - An *IouTransfer* that was part of the scenario during sandbox startup.
    - Bob's original *Iou* for $110.
    - 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 *DA 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 transac-


Template IouTrade:IouTrade@ffb9512d97abef53c681f67f4bc32a698c861dcbbd9c87f24c9b286fc2f000ec
buyer
Alice
seller
Bob
baseIouCid
#6:1
baseIssuer
EUR_Bank
baseCurrency
EUR
baseAmount
100.0
quoteIssuer
USD_Bank
quoteCurrency
USD
quoteAmount
110.0
Submit

tion graph, including who knows what, in *Test using scenarios* below.

1.2.5 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 instance 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 instance
additional parties, called observers, who can see the contract instance

For more information about the DA Ledger, consult *DA Ledger Model* for an in-depth technical description.

1.2.5.1 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:

```
daml 1.2
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 instance 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
          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
```

(continues on next page)

(continued from previous page)

```
        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*.

1.2.5.2 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:

```
iouTransferAliceCid <- submit eurBank do
  iouCid <- create Iou with
    issuer = eurBank
    owner = eurBank
    currency = "EUR"
    amount = 100.0
    observers = []
  exercise iouCid Iou_Transfer with newOwner = alice
```

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.

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

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	[]

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'
        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
```

(continues on next page)

(continued from previous page)

```
                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";
               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:
```

(continues on next page)

(continued from previous page)

```
        #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.

1.2.6 Integrate with the ledger

A distributed ledger only forms the core of a full DA Platform application.

To build automations and integrations around the ledger, the SDK has *language bindings* for the Ledger API in several programming languages.

To compile the Java integration for the quickstart application, run `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 that the Id of your new *AliceCoin* using step 1.. 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.ledger</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.
↪forHostWithLedgerIdDiscovery(ledgerhost, ledgerport, Optional.
↪empty());

// 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.

```
AtomicLong idCounter = new AtomicLong(0);
ConcurrentHashMap<Long, Iou> contracts = new ConcurrentHashMap<>();
BiMap<Long, Iou.ContractId> idMap = Maps.synchronizedBiMap(HashBiMap.
↪create());
```

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

```
client.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);

                    });
        });
```

Note the use of `blockingForEach` to ensure that the contract store is fully built and the ledger-offset up to which the ACS provides data is known before moving on.

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.

```
Disposable ignore = 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.

```
private static Empty submit(LedgerClient client, String party, Command
↪c) {
    return client.getCommandSubmissionClient().submit(
            UUID.randomUUID().toString(),
            "IouApp",
            UUID.randomUUID().toString(),
            party,
            Instant.EPOCH,
            Instant.EPOCH.plusSeconds(10),
            Collections.singletonList(c))
            .blockingGet();
}
```

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

Listing 1: ExerciseCommand

```
Iou.ContractId contractId = idMap.get(Long.parseLong(req.params("id
↪")));
ExerciseCommand exerciseCommand = contractId.exerciseIou_Transfer(m.
↪get("newOwner").toString());
```

Listing 2: CreateCommand

```
Iou iou = g.fromJson(req.body(), Iou.class);
CreateCommand iouCreate = iou.create();
```

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*.

1.2.7 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 and platform.

Chapter 2

Writing DAML

2.1 An introduction to DAML

DAML is a smart contract language designed to build composable applications on an abstract *DA 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 *DA Ledger Model* and how to use the DAML SDK 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 download the sources for section 1 into a folder called `1_Token`, run `daml new 1_Token daml-intro-1`.

Prerequisites:

> You have installed the DAML 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-transferrable 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

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 again.

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 of each DAML file is a pragma informing the compiler of the language version and the module name:

```
daml 1.2
module Token where
```

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

```
-- The first line of a DAML file is a pragma telling the compiler the
↪language
-- version to use.
daml 1.2
-- A DAML file defines a module. The second line of a DAML file gives the
-- module a name.
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 instance. These are the parties whose *authority* is required to create the contract or archive it again – 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 scenarios*, you'll learn about how to try out the `Token` contract template in DAML's inbuilt `scenario` testing language.

2.1.2 2 Testing templates using scenarios

In this section you will test the `Token` model from *1 Basic contracts* using DAML's inbuilt `scenario` language. You'll learn about the basic features of scenarios:

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

2.1.2.1 Scenario basics

A `Scenario` 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 some external information like party identities, and ledger time.

Below is a basic scenario that creates a `Token` for a party called Alice.

```
token_test_1 = scenario do
  alice <- getParty "Alice"
  submit alice do
    create Token with owner = alice
```

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

Before you can create any `Token` contracts, you need some parties on the test ledger. The above scenario uses the function `getParty` 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 `getParty` is an `Action` that can only be performed once the `Scenario` 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 the scenario is running on, `getParty` may have to look up a party identity or create a new party.
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 `getParty` 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: a `Party` and an `Update`.

Just like `Scenario` is a recipe for a test, `Update` is a recipe for a transaction. `create Token with owner = alice` is an `Update`, which translates to the transaction creating a `Token` with owner Alice.

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

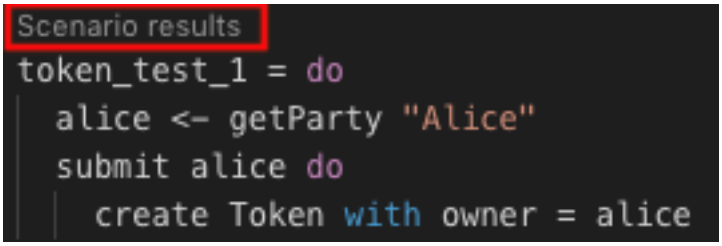
You could write this as `submit alice (create Token with owner = alice)`, but just like scenarios, you can assemble updates using `do` blocks. A `do` block always takes the value of the last statement within it so the syntax shown in the scenario above gives the same result, whilst being easier to read.

2.1.2.2 Running scenarios

There are two ways to run scenarios:

In DAML Studio, providing visualizations of the resulting ledger
Using the command line, useful for continuous integration

In DAML Studio, you should see the text Scenario results just above the line `token_test_1 = do`. Click on it to display the outcome of the scenario.



This opens the scenario 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 scenario, 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 scenario, 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 = scenario do
  alice <- getParty "Alice"
  bob <- getParty "Bob"
  submit alice do
    create Token with owner = bob
  submit bob do
    create Token with owner = alice
```

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

```
Scenario execution failed on commit at Intro_2_Scenario:70:3:
  #0: create of Intro_2_Scenario:Token at DA.Internal.Prelude:377:26
      failed due to a missing authorization from 'Bob'

Last source location: DA.Internal.Prelude:377:26

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

Partial transaction:
  Sub-transactions:
    #0
    └> create Intro_2_Scenario:Token
       with
         owner = 'Bob'
```

The scenario failed, as expected, but scenarios 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 scenario running thereafter, or fail if the submission succeeds, you can use the `submitMustFail` function:

```
token_test_2 = scenario do
  alice <- getParty "Alice"
  bob <- getParty "Bob"

  submitMustFail alice do
    create Token with owner = bob
  submitMustFail bob do
    create Token with owner = alice
```

(continues on next page)

(continued from previous page)

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

`submitMustFail` never has an impact on the ledger so the resulting tabular scenario 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 `archive` instead of `create`. Where `create` takes an instance of a template, `archive` 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 `archive` the Token Alice has created, you need to get a handle on its contract ID. In scenarios, 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 scenario first checks that Bob cannot archive Alice's Token and then Alice successfully archives it:

```
token_test_3 = scenario do
  alice <- getParty "Alice"
  bob <- getParty "Bob"

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

  submitMustFail bob do
    archive alice_token

  submit alice do
    archive alice_token
```

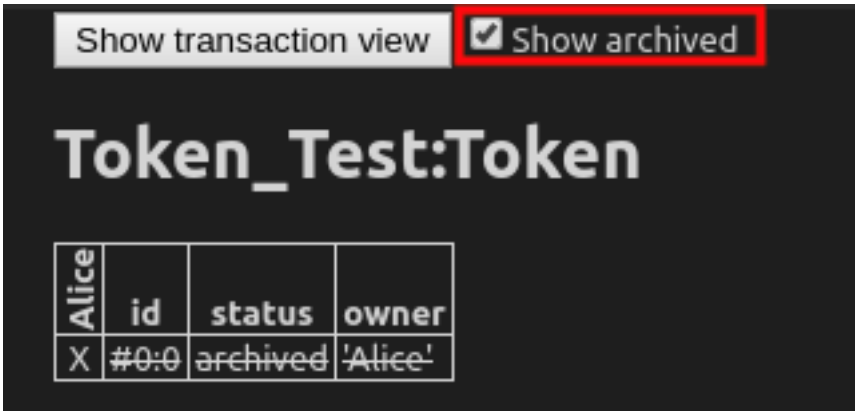
2.1.2.5 Exploring the ledger

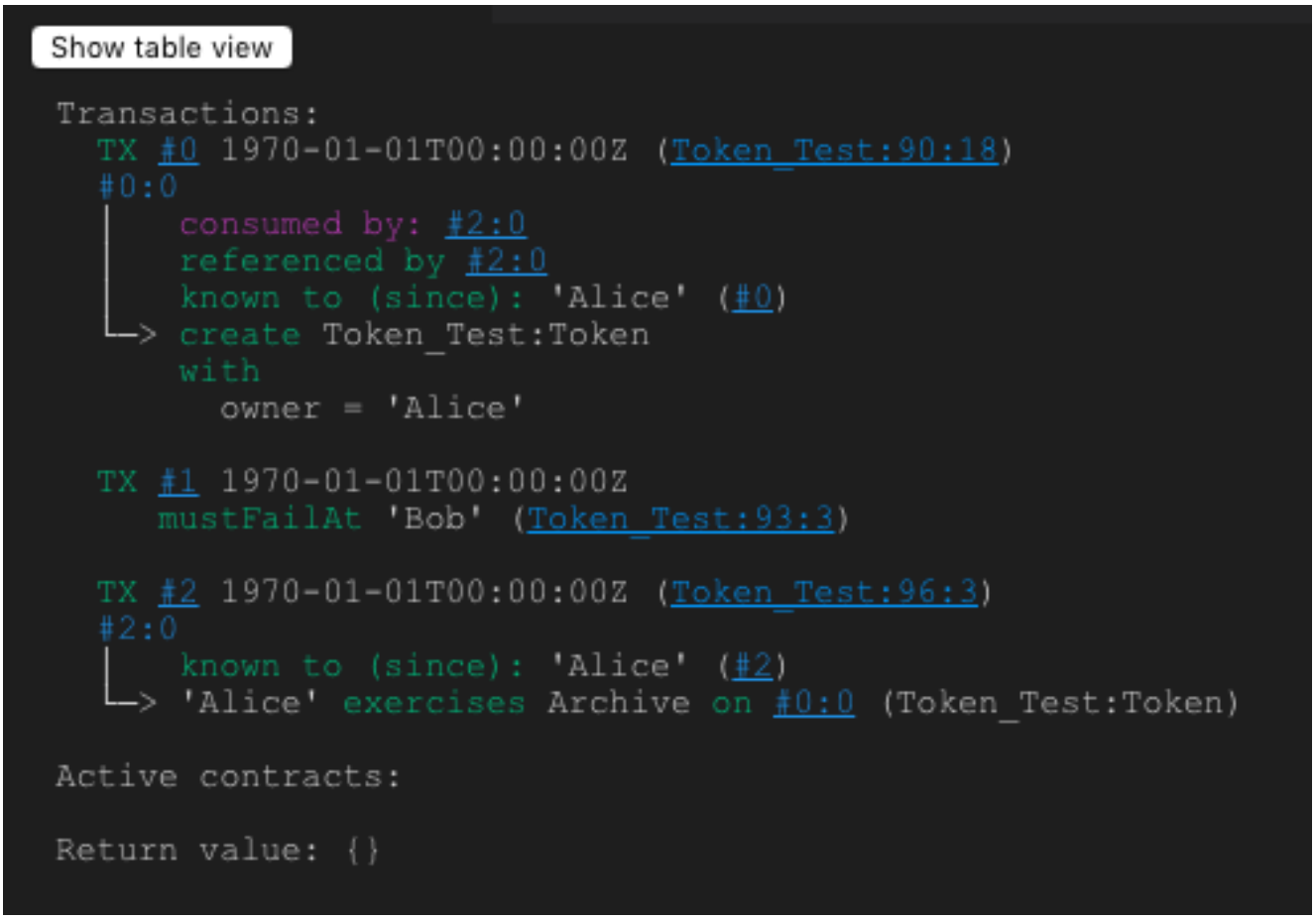
The resulting scenario view is empty, because there are no contracts left on the ledger. However, if you want to see the history of the ledger, e.g. to see how you got to that state, tick the Show archived box at the top of the ledger view:

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 scenario 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 scenario.


Show transaction view
Show archived
Token_Test:Token
Alice
id
status
owner
X
#0:0
archived
'Alice'


Show table view
Transactions:
TX #0 1970-01-01T00:00:00Z (Token_Test:90:18)
#0:0
consumed by: #2:0
referenced by #2:0
known to (since): 'Alice' (#0)
create Token_Test:Token
with
owner = 'Alice'
TX #1 1970-01-01T00:00:00Z
mustFailAt 'Bob' (Token_Test:93:3)
TX #2 1970-01-01T00:00:00Z (Token_Test:96:3)
#2:0
known to (since): 'Alice' (#2)
'Alice' exercises Archive on #0:0 (Token_Test:Token)
Active contracts:
Return value: {}

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 scenario 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 scenarios, try the following exercises:

1. Write a template for a second type of Token.
2. Write a scenario 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.

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 scenarios*, you learnt about the scenario 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 instance 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.

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 scenarios*. 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 Oblique.
`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 scenario instantiates each one of these types, manipulates it where appropriate, and tests the result.

```
import DA.Time
import DA.Date

native_test = scenario do
  alice <- getParty "Alice"
  bob <- getParty "Bob"
  let
    my_int = -123
    my_dec = 0.001
    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 (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 scenario:

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. More on packages, imports and the standard library later.
Most of the variables are declared inside a `let` block.
That's because the `scenario do` block expects scenario actions like `submit` or `getParty`. 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.
None of the variables 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. And you can always choose to add them 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 scenario and see what happens to the scenario 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 = scenario do
  accountant <- getParty "Bob"
  alice <- getParty "Alice"
  bob <- getParty "Bank of Bob"

  submit accountant do
    create 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

tuple_test = scenario do
  let
    my_key_value = ("Key", 1)
    my_coordinate = (1.0, 2.0, 3.0)

  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 scenario instantiates a few lists of integers and demonstrates the most important list functions.

```
import DA.List

list_test = scenario do
  let
    empty : [Int] = []
```

(continues on next page)

(continued from previous page)

```
    one = [1]
    two = [2]
    many = [3, 4, 5]

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

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

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

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

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

Records

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

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

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

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

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

record_test = scenario do
```

(continues on next page)

(continued from previous page)

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

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

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

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

    -- 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 scenario, 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
```

(continues on next page)

(continued from previous page)

```
  my_int : Int
  my_dec : Decimal
  my_list : [Text]
    deriving (Eq)

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

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

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

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

`Eq` is what is called a *type-class*. You can think of a type-class 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 type-classes 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.

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)
```

(continues on next page)

(continued from previous page)

```
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 = scenario do
  accountant <- getParty "Bob"
  owner <- getParty "Alice"
  bank_party <- getParty "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
    create CashBalance with
      accountant
      cash
      account
```

If you look at the resulting scenario 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
```

(continues on next page)

(continued from previous page)

```
    deriving (Eq, Show)

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

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

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

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

cash_balance_test = scenario do
  accountant <- getParty "Bob"
  owner <- getParty "Alice"
  bank_party <- getParty "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
    create CashBalance with
      accountant
      owner
      cash
      location = InHand
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

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

variant_access_test = scenario 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.
```

(continues on next page)

(continued from previous page)

```
    l_value = case l of
      Left i -> i
      Right i -> error "Expecting Left"
    -- Comment out at your own peril
    {-
    r_value = case r of
      Left i -> i
      Right i -> error "Expecting Left"
    -}

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

    -- If we don't care about values or even constructors, we can use
↪wildcards
    l_value2 = case l of
      Left i -> i
      Right _ -> error "Expecting Left"
    l_value3 = case l of
      Left i -> i
      _ -> error "Expecting Left"

    day = Sunday
    weekend = case day of
      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 = scenario do
  let
    eq_record = EqRecord with
      my_txt = "Text"
      my_int = 2
      my_dec = 2.5
      my_list = ["One", "Two", "Three"]

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

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

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

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

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

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

Throughout the scenario, `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
    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 = scenario do
  accountant <- getParty "Bob"
  owner <- getParty "Alice"
  bank_party <- getParty "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
     create Account with
      accountant
      owner
      bank
      number = "ABC123"
```

(continues on next page)

(continued from previous page)

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

  -- Now the accountant updates the telephone number for the bank on the
↪account
  new_account <- submit accountant do
    account <- fetch accountCid
    archive accountCid
    create account with
      bank = account.bank with
        telephone = "098 7654 321"

  -- The `account` field on the balance now refers to the archived
  -- contract, so this will fail.
  submitMustFail accountant do
    balance <- fetch balanceCid
    fetch balance.account
```

The scenario above uses the `fetch` function, which retrieves the arguments of an active contract using its contract ID.

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 fetches the arguments of the old account, 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
```

(continues on next page)

(continued from previous page)

```
    number : Text
    bank : Bank
  where
    signatory accountant

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

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

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

id_ref_test = scenario do
  accountant <- getParty "Bob"
  owner <- getParty "Alice"
  bank_party <- getParty "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
     create Account with
      accountant
      owner
      bank
      number = "ABC123"

  balanceCid <- submit accountant do
    account <- fetch accountCid
    create CashBalance with
      accountant
```

(continues on next page)

(continued from previous page)

```
      cash
      account = key account

  -- Now the accountant updates the telephone number for the bank on the
↪account
  new_accountCid <- submit accountant do
    account <- fetch accountCid
    archive accountCid
    create account with
      bank = account.bank with
        telephone = "098 7654 321"

  -- Thanks to contract keys, the current account contract is fetched
  submit accountant do
    balance <- fetch balanceCid
    (cid, account) <- fetchByKey @Account balance.account
    assert (cid == new_accountCid)
```

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 maintainters are part of the key is ensured by only having the variable *key*

Note how the `fetch` in the final `submit` block has become a `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.

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.

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.

If you paid a lot of attention in *3 Data types*, you may have noticed that the `create` statement 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 conveneience. 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 scenario:

```
choice_test = scenario do
  owner <- getParty "Alice"
  party <- getParty "Bob"

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

  -- The bank can't change its own telephone number as the accountant
↪controls
  -- that choice.
  submitMustFail party do
    exercise contactCid UpdateTelephone with
      newTelephone = "098 7654 321"

  new_contact_Cid <- submit owner do
    exercise contactCid UpdateTelephone with
      newTelephone = "098 7654 321"

  submit owner do
    new_contact <- fetch new_contact_Cid
    assert (new_contact.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.

`exercise` returns an `Update r` where `r` is the return type specified on the choice, allowing the new `ContractId Contact` to be stored in the variable `new_contactCid`.

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 scenario:

```
    controller party can
      UpdateAddress
        : ContractId Contact
        with
```

(continues on next page)

(continued from previous page)

```
          newAddress : Text
        do
          create this with
            address = newAddress
```

```
  new_contact_Cid <- submit party do
    exercise new_contact_Cid UpdateAddress with
      newAddress = "1-10 Bobstreet"

  submit owner do
    new_contact <- fetch new_contact_Cid
    assert (new_contact.address == "1-10 Bobstreet")
```

If you open the scenario 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 three kinds of action: `create`, `exercise` and `fetch`. All actions are performed on a contract.

> A `create` action contains the contract arguments and changes the contract's status from *non-existent* to *active*.
> A `fetch` action checks the existence and activeness of a contract.
> An `exercise` action contains the choice arguments and a transaction 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*.

The consequences of exercise nodes turn each transaction into an ordered tree of (sub-) transactions, or, equivalently, a forest of actions. Actions are in one-to-one correspondence with proper sub-transactions. You can see the action and their consequences in the transaction view of the above scenario:

```
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)
```

(continues on next page)

(continued from previous page)

```
  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:
      #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 scenario. 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) 2019 The DAML Authors. All rights reserved.
-- SPDX-License-Identifier: Apache-2.0

daml 1.2
module SimpleIou where

data Cash = Cash with
  currency : Text
  amount : Decimal
    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 = scenario do
  alice <- getParty "Alice"
  bob <- getParty "Bob"
  charlie <- getParty "Charlie"
  dora <- getParty "Dora"

  -- The bank issues an Iou for $100 to alice
  iou <- submit dora do
```

(continues on next page)

(continued from previous page)

```
    create SimpleIou with
      issuer = dora
      owner = alice
      cash = Cash with
        amount = 100.0
        currency = "USD"

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

  -- Bob transfers it to Charlie
  submit bob do
    exercise 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 `Scenario` 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 scenarios.

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
```

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

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

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

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

  -- But positive Ious still work, of course
```

(continues on next page)

(continued from previous page)

```
iou <- submit dora do
  create 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 scenarios.

`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
  exercise iou Transfer with
    newOwner = alice

-- but can transfer to bob
iou2 <- submit alice do
  exercise 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
```

(continues on next page)

(continued from previous page)

```
            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
  passToDate (date 2019 Jun 1)
  -- Bob cannot redeem on a Saturday
  submitMustFail bob do
    exercise iou2 Redeem

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

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

  -- Bob can redeem 8am on Monday.
  pass (hours 2)
  submit bob do
    exercise 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 effective time (LET)* and the *record time (RT)*. The ledger effective time is set by the submitter of a transaction, the record time is set by the consensus protocol.

Each DAML ledger has a policy on the allowed difference between LET and RT called the *skew*. The submitter 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 LET 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.

Suppose now that the ledger had a skew of 10 seconds, but a submission took less than 4 seconds to commit. At 18:00:05, Alice could submit a transaction with a LET of 17:59:59 to redeem an Iou. It would be a valid transaction and be committed successfully as `getTime` will return 17:59:59 so `hrs == 17`. Since RT will be before 18:00:09, `LET - 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.

Time in scenarios

In scenarios, record and ledger effective time are always equal. You can set them using the following functions:

> `passToDate`, which takes a date sets the time to midnight (UTC) of that date and `pass`
> `pass`, 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 effective time or relative time are strictly increasing. The only guarantee is that ledger effective time is increasing with causality. That is, if a transaction `TX2` depends on a transaction `TX1`, then the ledger enforces that the LET of `TX2` is greater than or equal to that of `TX1`:

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

pass (days (-3))
submitMustFail alice do
  exercise iou3 Redeem
```

2.1.5.4 Actions and `do` blocks

You have come across `do` blocks and `<-` notations in two contexts by now: `Scenario` 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:

`getTime` is a good example of 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 effective 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 `Scenario` 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 `Scenario 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`, `getParty party`, `pass time`, `submit party update`, `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`.
> `pass (days 10) : Scenario ()` is a recipe for a transaction that doesn't submit any transactions, but has the side-effect of changing the LET 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 (create iou) : Scenario (ContractId Iou)` is a recipe for a scenario in which Alice evaluates the result of `create iou` to get a transaction and a return value of type `ContractId Iou`, and then submits that transaction to the ledger.

Any DAML ledger knows how to perform actions of type `Update a`. Only some know how to run scenarios, meaning they can perform actions of type `Scenario a`.

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 scenario is a list of interactions with the ledger (`submit`, `getParty`, `pass`, etc). So a scenario followed by another scenario is again a scenario.

This is where `do` blocks come in. `do` blocks allow you to combine small actions in to bigger actions, using the results of earlier actions in later ones.

```
sub_scenario1 : Scenario (ContractId SimpleIou) = scenario do
  alice <- getParty "Alice"
  dora <- getParty "Dora"

  submit dora do
    create SimpleIou with
      issuer = dora
      owner = alice
      cash = Cash with
        amount = 100.0
        currency = "USD"

sub_scenario2 : Scenario Int = scenario do
  getParty "Nobody"
  pass (days 1)
  pass (days (-1))
  return 42

sub_scenario3 : Scenario (ContractId SimpleIou) = scenario do
  bob <- getParty "Bob"
  dora <- getParty "Dora"

  submit dora do
    create SimpleIou with
      issuer = dora
      owner = bob
      cash = Cash with
        amount = 100.0
        currency = "USD"

main_scenario : Scenario () = scenario do
  dora <- getParty "Dora"

  iou1 <- sub_scenario1
  sub_scenario2
  iou2 <- sub_scenario3

  submit dora do
    archive iou1
```

(continues on next page)

(continued from previous page)

```
    archive iou2
```

Above, we see `do` blocks in action for both `Scenario` 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_scenario2 : Scenario Int`.

2.1.5.5 Failing actions

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

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

Transactions and scenarios succeed or fail *atomically* as a whole. So an occurrence of an `abort` action will always fail the **entire** evaluation of the current `Scenario` or `Update`.

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 RNGs state. Based on the `Heads` and `Tails` results, a return value of type `a` is calulated.

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

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

flipCoin : CoinGame Face
getCoin : Scenario 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 `Scenario` 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 = scenario do
  -- the coin is pseudo-random on LET so change the parameter to change the
↪game
  passToDate (date 2019 Jun 1)
  pass (seconds 2)
  coin <- getCoin
  let
    game = do
      f1r <- flipCoin
      f2r <- flipCoin
      f3r <- flipCoin

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

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

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

In a `Scenario` context you can get a `Coin` using the `getCoin` action, which uses the LET 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. For example:

> 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 Lipovaa)
> Programming in Haskell (Graham Hutton)
> Real World Haskell (Bryan O'Sullivan, Don Stewart, John Goerzen)

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 scenario succeeds or fails depending on the value of `abortScenario`:

```
nonPerformedAbort = scenario do
  let abortScenario = False
  let failingAction : Scenario () = abort "Foo"
  let successfulAction : Scenario () = return ()
  if abortScenario 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 x y
 | y == 0 = Some 1
 | y > 0 = let Some z = optPow x (y - 1)
           in Some (y * z)
 | otherwise = None
```

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

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

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

```
nonPerformedError = scenario 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

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 again after receiving the goods. Bob would have a record such actions, but would have to resort to off-ledger means to get his money back.

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

```
simple_iou_test = scenario do
  alice <- getParty "Alice"
  bob <- getParty "Bob"

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

  pass (days 1)
  -- Bob delivers the goods

  pass (minutes 10)
  -- Alice just deletes the payment again.
  submit alice do
    archive 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
```

(continues on next page)

(continued from previous page)

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

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

```
iou_test = scenario do
  alice <- getParty "Alice"
  bob <- getParty "Bob"

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

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

  -- But can't transfer it to Bob
  submitMustFail alice do
    exercise 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 buildng 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

    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
    create IouProposal with
      iou = Iou with
        issuer = alice
        owner = bob
        cash = Cash with
          amount = 100.0
          currency = "USD"

  submit bob do
    exercise iouProposal IouProposal_Accept
```

The `IouProposal` contract carries the authorithy 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
```

(continues on next page)

(continued from previous page)

```
        do
          create iou

    controller newOwner can
      IouTransferProposal_Reject
        : ContractId Iou
        do
          create iou

      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 <- getParty "Charlie"

  -- Alice issues an iou using a transfer proposal
  tpab <- submit alice do
    create IouTransferProposal with
      newOwner = bob
      iou = Iou with
        issuer = alice
        owner = alice
        cash = Cash with
```

(continues on next page)

(continued from previous page)

```
        amount = 100.0
        currency = "USD"

  -- Bob accepts
  iou2 <- submit bob do
    exercise tpab IouTransferProposal_Accept

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

  -- Charlie accepts
  submit charlie do
    exercise 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 scenrario 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 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
```

(continues on next page)

(continued from previous page)

```
  where
    signatory receiver

    controller sender can
      nonconsuming Send_Iou
        : ContractId Iou
        with
          iouCid : ContractId Iou
        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 a 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
    create IouSender with
      sender = alice
      receiver = bob

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

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

  -- Bob sends it on to charlie
  submit bob do
    exercise 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

The final transaction in the scenario of the the source file for this section 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_Exercise` 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:
```

(continues on next page)

(continued from previous page)

```
    #12:1
    │   known to (since): 'Bob' (#12), 'Charlie' (#12)
    └─> fetch #11:3 (Parties:Iou)

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

Note that authority is not automatically transferred transitively.

```
template NonTransitive
  with
    partyA : Party
    partyB : Party
  where
    signatory partyA

    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
    create NonTransitive with
      partyA = alice
      partyB = bob
```

(continues on next page)

(continued from previous page)

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

  submitMustFail bob do
    exercise 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 finally put everything you have learned together to build a simple asset holding and trading model akin to that in the *Quickstart guide*. 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 *Quickstart guide*. 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.

2.1.7.1 DAML projects

DAML is organized in packages and modules. A DAML project is specified using a single `daml.yaml` file, and compiles into a package. Each DAML file within a project becomes a DAML module. You can start a new project with a skeleton structure using `daml new project_name` in the terminal.

Each DAML project has a main source file, which is the entry point for the compiler. A common pattern is to have a main file called `LibraryModules.daml`, which simply lists all the other modules to include.

A minimal project would contain just two files: `daml.yaml` and `daml/LibraryModules.daml`. Take a look at the `daml.yaml` for this project:

```
sdk-version: __VERSION__
name: __PROJECT_NAME__
source: daml/LibraryModules.daml
```

(continues on next page)

(continued from previous page)

```
version: 1.0.0
dependencies:
  - daml-prim
  - daml-stdlib
```

You can generally set `name` and `version` freely to describe your project. `dependencies` lists package dependencies: you should always include `daml-prim`, and `daml-stdlib` gives access to the DAML standard library.

You compile a DAML project by running `daml build` from the project root directory. This creates a `dar` package in `dist/project_name.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 contract model.

2.1.7.2 Project structure

This project contains an asset holding model for transferrable, 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, and the last one to a file name. The folder structure therefore looks like this:

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

Each file contains the DAML pragma and module header. For example, `daml/Intro/Asset/Role.daml`:

```
daml 1.2
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 ()
import Intro.Asset.Role ()
```

(continues on next page)

(continued from previous page)

```
import Intro.Asset.Trade ()

import Test.Intro.Asset ()
import Test.Intro.Asset.Role ()
import Test.Intro.Asset.Trade ()
```

Imports always have to appear just below the module declaration. The `()` behind each `import` above is optional, and lets you only import selected names.

In this case, it suppresses an unused import warning. `LibraryModules` is not actually using any of the imports in `LibraryModules`. The `()` tells the compiler that this is intentional.

A more typical import statement is `import Intro.Asset` as found in `Test.Intro.Asset`.

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 scenarios

This project showcases how you can put the `Update` and `Scenario` actions you learnt about in 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
```

(continues on next page)

(continued from previous page)

```
        with
          otherCid : ContractId Asset
        do
          other <- fetch otherCid
          assertMsg
            "Merge failed: issuer does not match"
            (issuer == other.issuer)
          assertMsg
            "Merge failed: owner does not match"
            (owner == other.owner)
          assertMsg
            "Merge failed: symbol does not match"
            (symbol == other.symbol)
          archive otherCid
          create this with
            quantity = quantity + other.quantity
```

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

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

```
      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
```

(continues on next page)

(continued from previous page)

```
        return (transferredBaseCid, transferredQuoteCid)
```

The resulting transaction, with its two nested levels of consequences, can be seen in the `test_trade` scenario in `Test.Intro.Asset.Trade`:

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

(continues on next page)

(continued from previous page)

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

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

        #15:10
        │   referenced by #16:0
        │   known to (since): 'Alice' (#15), 'EUR_Bank' (#15), 'Bob' (#15)
        └─> create Intro.Asset:Asset
            with
              issuer = 'EUR_Bank'; owner = 'Alice'; symbol = "EUR";
↪quantity = 90.0; observers = []
```

Similar to choices, you can see how the scenarios in this project are built up from each other:

```
test_issuance = scenario do
  setupResult@(alice, bob, bank, aha, ahb) <- setupRoles

  assetCid <- submit bank do
    exercise aha Issue_Asset
      with
        symbol = "USD"
        quantity = 100.0

  submit bank do
    asset <- fetch assetCid
    assert (asset == Asset with
      issuer = bank
      owner = alice
      symbol = "USD"
      quantity = 100.0
      observers = []
        )

  return (setupResult, assetCid)
```

In the above, the `test_issuance` scenario in `Test.Intro.Asset.Role` uses the output of the `setupRoles` scenario in the same module.

The same line shows a new kind of pattern matching. Rather than writing `setupResults <- setupRoles` and then accessing the components of `setupResults` using `_1`, `_2`, etc., you can give them names. It's equivelent to writing

```
setupResults <- setupRoles
case setupResults of
  (alice, bob, bank, aha, ahb) -> ...
```

Just writing `(alice, bob, bank, aha, ahb) <- setupRoles` would also be legal, but `setupResults` 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:

1. A party submits a transaction. Remember, a transaction is just a list of actions.
2. The transaction is interpreted, meaning the `Update` corresponding to each action is evaluated in the context of the ledger to calculate all consequeces, including transitive ones (consequences of consequences, etc).
3. The views of the transaction that parties get to see (see *Privacy*) are calculated in a process called *blinding*, or *projecting*.
4. The blinded views are distributed to the parties.
5. The transaction is *validated* based on the blinded views and a consensus protocol depending on the underlying infrastructure.
6. If validation succeeds, the transaction is *committed*.

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, due to 2., is that the submitter of a transaction knows all consequences of their submitted transaction – there are no surprises in DAML. However, it also means that the submitter must have all the information to interpret the transaction.

That's also imprtant 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 proposeing the trade. You can try out what happens if she didn't do that by removing that transaction.

```
  usdCid <- submit alice do
    exercise 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 couterparties, 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:

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

Item 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 on 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 scenario 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` scenario, 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 disclosues is often called divulgence and needs to be considered when designing DAML models for privacy sensitive applications.

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
```

(continues on next page)

(continued from previous page)

```
    -- 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:
signatories `signatory` keyword
Required. The parties (see the *Party* type) who must consent to the creation of an instance of this contract. You won't be able to create an instance of 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 an instance of this contract 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
```

(continues on next page)

(continued from previous page)

```
    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.

consumability `nonconsuming` keyword
: By default, contracts are archived when a choice on them is exercised, which means that choices can no longer be exercised on them. If you include `nonconsuming`, this choice can be exercised over and over.

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 instance 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 instance 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 effective 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 instance 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 instance 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 Signatory parties

```
  where
    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 an instance 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 instance is visible to all signatories (as well as the other stakeholders of the contract). That is, the compiler automatically adds signatories as observers.
You **must** have least one signatory per template. You can have many, either as a comma-separated list or reusing the keyword. You could pass in a list (of type `[Party]`).

2.2.2.4 Observers

```
    observer exampleParty2
```

`observer` keyword. After `where`. Followed by at least one `Party`.
Observers are additional stakeholders, so the contract instance 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.

2.2.2.5 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
```

(continues on next page)

(continued from previous page)

```
      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.

2.2.2.6 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 `<>`.

2.2.2.7 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.

2.2.2.8 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 an instance of this contract 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:

`choice` first or `controller` first
Choice name
Controllers
- *Non-consuming choices*
- *Return type*

Choice arguments
Choice body

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
      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.

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.

Non-consuming choices

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

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

Listing 7: 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.
By default, choices are *consuming*: when a choice on a contract is exercised, that contract instance is *archived*. Archived means that it's permanently marked as being inactive, and no more choices can be exercised on it, though it still exists on the ledger.

This is 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.

2.2.3.4 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.

2.2.3.5 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:

Background
Binding variables
do
create
exercise
exerciseByKey
fetch
fetchByKey
lookupByKey
abort
assert
getTime
return
let
this

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 an instance of that 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 instance 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 instance with that *contract key*, instead of the contract ID.
As well as the authorization that `fetch` requires, you also need authorization from one of the `maintainers` of the key.

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 it does exist, `lookupByKey` returns the `ContractId` of the contract; otherwise, it returns `None`. If it returns `None`, this guarantees that no contract has this key. This does **not** cause the transaction to abort.
All of the maintainers of the key must authorize the lookup (by either being signatories or by submitting the command to lookup), otherwise this will fail.

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 effective 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:

- *Built-in types*
  - *Table of built-in primitive types*
  - *Escaping characters*
  - *Time*
- *Lists*
  - *Summing a list*
- *Records and record types*
  - *Data constructors*
  - *Accessing record fields*
  - *Updating record fields*
  - *Parameterized data types*
- *Type synonyms*
  - *Function types*
- *Algebraic data types*
  - *Product types*
  - *Sum types*
  - *Pattern matching*

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`	fixed point decimals	`1.0`	`Decimal` values are rational numbers with precision 38 and scale 10: numbers of the form `x / 10^10` where x is an integer with `|x| < 10^38`.
`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)
```

(continues on next page)

(continued from previous page)

```
-- 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`, 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.

A very useful sum type is `data Optional a = None | Some a`. 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"
    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:

Working with time
Working with numbers
Working with text
Working with lists
- *Folding*

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 *scenario tests* only) use `pass : RelTime -> Scenario Time` to advance the ledger effective 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*:

Definitions
- *Values*
- *Functions*

Arithmetic operators
Comparison operators
Logical operators
If-then-else
Let

2.2.7.1 Definitions

Use assignement 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 contract instances* 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:

Defining functions
Partial application
Functions are values
Generic functions

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: `Decmial -> (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 == 3)
    assert (round 3.5 == 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:

Scenario keyword
Submit
submitMustFail
Scenario body
- *Updates*
- *Passing time*
- *Binding variables*

For an introduction to scenarios, see *Testing using scenarios*.

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)

    -- 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*:

File structure
Imports
Libraries
Comments
Contract identifiers

2.2.10.1 File structure

Language version (`daml 1.2`).
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 looks different to one on the DA Platform.

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:

DAML archives
Importing DAML archives

2.2.11.1 DAML archives

When a DAML project is build with `daml build`, build artifacts are generated in the hidden directory `.daml/dist/` relative to the project root directory. The main build artifact of a project is the *DAML archive*, recognized by the `.dar` file ending. DAML archives are platform independent. They can be deployed on a ledger (see *deploy*) or can be imported into other projects as a package dependency.

2.2.11.2 Importing DAML archives

A DAML project can import DAML archive dependencies. Note that currently there is no tooling for DAML package management. To import a package `Bar` in project `Foo`, add the file path of the `Bar` DAML archive to the `dependencies` section of the *daml.yaml* project file:

```yaml
sdk-version: 0.0.0
name: foo
source: daml
version: 1.0.0
exposed-modules:
  - Some.Module
  - Some.Other.Module
dependencies:
  - daml-prim
  - daml-stdlib
  - /home/johndoe/bar/.daml/dist/bar.dar
```

The import path needs to be the relative or absolute path pointing to the created DAML archive of the `bar` project. The archive can reside anywhere on the local file system. Note that the SDK versions of the packages `foo` and `bar` need to match, i.e. it is an error to import a package that was created with an older SDK.

Once a package has been added to the dependencies of the `foo` project, modules of `bar` can be imported as usual with the `import Some.Module` directive (see *Imports*). If both projects `foo` as well as `bar` contain a module with the same name, the import can be disambiguated by adding the package name in front of the module name, e.g. `import "bar" Some.Module`.

Note that all modules of package `foo` that should be available as imports of other packages need to be exposed by adding them to the `exposed-modules` stanza of the *daml.yaml* file. If the `exposed-modules` stanza is omitted, all modules of the project are exposed by default.

2.2.12 Contract keys

Contract keys are an optional addition to templates. They let you specify a way of uniquely identifying contract instances, 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 expression but 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. 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. Maintainers are the parties that know about all of the keys that they are party to, and are used by the engine to guarantee uniqueness of contract keys. The maintainers **must** be signatories of the contract.

Keys are unique to their maintainers. For example, say you have a key that you're using as the identifer for a `BankAccount` contract. You might have `key (bank, accountId) : (Party, Text)`. When you create a new bank account, the contract key ensures that no-one else can have an account with the same `accountID` at that bank. But that doesn't apply to other banks: for a contract with a different bank as maintainer, you could happily re-use that `accountID`.

When you're 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.

Checking of the keys is done automatically at execution time, by the DAML exeuction 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 keys functions

Contract keys introduce several new functions.

`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 the currently-used `fetch`.

It returns a tuple of the ID and the contract object (containing all its data).

You need authorization from **at least one** of the maintainers to run `fetchByKey`. A maintainer can authorize by being a signatory, or by submitting the command/being a controller for the choice.

`fetchByKey` fails and aborts the transaction if:

> you don't have sufficient authorization
> you're not a stakeholder of the contract you're trying to fetch

This means that if it fails, it doesn't guarantee that a contract with that key doesn't exist, just that you can't see one.

Moreover, future versions of DAML will enforce that when using `fetchByKey` the submitter of the transaction is one of the maintainers. It's therefore advised to write your contract key workflows with this future limitation in mind.

Because different templates can use the same key type, you need to specify the type of the contract you are trying to fetch using the `@ContractType` syntax.

`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`.

You need authorization from **all** of the maintainers to run `lookupByKey`, and it can only be submitted by one of the maintainers.

If the lookup fails (ie returns `None`), this guarantees that no contract has this key.

Unlike `fetchByKey`, the transaction **does not fail** if a contract with the key doesn't exist: instead, `lookupByKey` just returns `None`.

To get the data from the contract once you've confirmed it exists, you'll still need to use `fetch`.

Moreover, like `fetchByKey`, future versions of DAML will enforce the submitter of the transaction is one of the maintainers. It's therefore advised to write your contract key workflows with this future limitation in mind.

Because different templates can use the same key type, you need to specify the type of the contract you are trying to fetch using the `@ContractType` syntax.

`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 of the key maintainers.

Because different templates can use the same key type, you need to specify the type of the contract you are trying to fetch using the `@ContractType` syntax.

2.2.13 Generic Templates

Sometimes different DAML templates have a common structure. Typically this occurs when there is some logic (usually in choices) that can be applied to many different underlying contracts. Generic templates allow you to capture this logic in a single place, instead of having to duplicate it for every template. Let's see a simple example to understand what this means.

2.2.13.1 Example: Generic Proposal

Suppose we want to model a propose and accept workflow. This means that a party can propose a contract to a specific party, who may accept it with the terms of that contract. We see this pattern occur frequently for different types of contracts. Of course, we could implement one proposal template for every underlying template. However this is tedious and error prone. Alternatively, we can write it once and for all using a generic template.

This is how the generic `Proposal` template looks in DAML.

```
template Template t => Proposal t
  with
    asset : t
    receiver : Party
  where
    signatory (signatory asset \\ [receiver])

    controller receiver can
      Accept : ContractId t
        do create asset
```

There are several things to notice in this short template.

Firstly, where we usually see a template name, there is now a more general *template header* `Template t => Proposal t`. This can include multiple type parameters and constraints on those types. In this case `Proposal` takes a single type parameter `t` representing the type of the underlying asset. The `Template` constraint says that `t` is not just any type but a contract template with signatories, choices and more.

Secondly, the `asset` parameter to the template has the abstract type `t`. We don't know anything about `asset` other than that it is an instance of `Template`. However this is all we need to implement the proposal template.

Notice that the signatories of the proposal are obtained from the signatories of the underlying contract. This is done by calling the overloaded `signatory` method. In this case we exclude the receiving party, as this is the one whose authorization we want to gain from accepting the proposal.

Finally let's look at the `Accept` choice which characterizes the propose and accept workflow. The receiver can `Accept` which results in a contract of the underlying asset type being created. We are able to call `create` on the asset since we know it satisfies the `Template` constraint.

2.2.13.2 Template Instances

The above template soundly represents the proposal workflow, but we have not yet used it on a concrete (non-generic) template. We call the concrete instantiation a *template instance*.

Let's introduce a very simple `Coin` contract that we can use in the proposal.

```
template Coin
  with
    issuer : Party
    owner : Party
  where
    signatory issuer, owner
```

We would like to model an issuing party (e.g. a bank) proposing a coin contract for an individual to accept. To do this, we need to explicitly state our intention to use `Coin` in a `Proposal`. We do this using the `template instance` syntax.

```
template instance CoinProposal = Proposal Coin
```

Note that we must choose a name, here `CoinProposal`, for the template instance. It serves two purposes. First, it provides a type synonym for the underlying type, here `Proposal Coin`. Second, it

must be used to create contracts of this template in some client languages (for example when using the Java ledger bindings).

With a template instance in place, we can create and exercise choices on contracts of this type.

```
coinIssuance = scenario do
  alice <- getParty "alice"
  bank <- getParty "bank"
  let coin = Coin with issuer = bank; owner = alice
  let proposal = Proposal with asset = coin; receiver = alice
  propId <- bank `submit` create proposal
  alice `submit` exercise propId Accept
```

We construct the underlying asset and the proposal data using the `Coin` and `Proposal` data constructors respectively. The bank is able to create the coin proposal which Alice can then accept. This scenario results in two contract creations: first the proposal contract, which is consumed to give rise to the coin contract.

2.3 DAML Studio

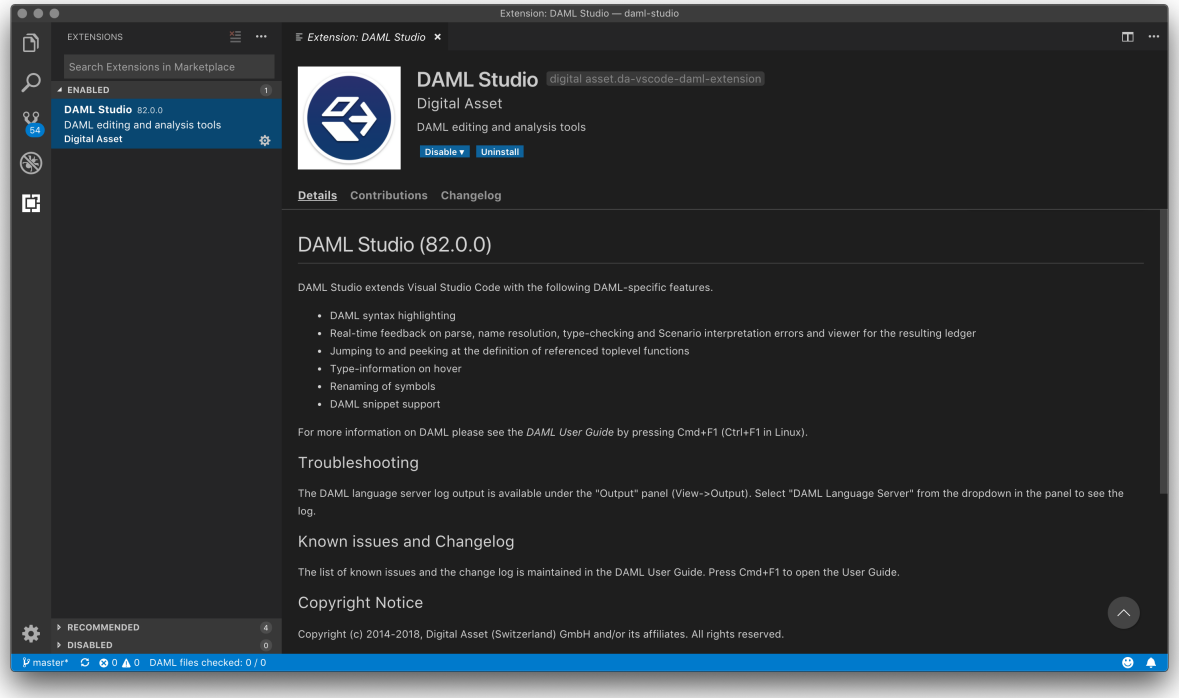
DAML Studio is an integrated development environment (IDE) for DAML. It is an extension on top of Visual Studio Code (VS Code), a cross-platform, open-source editor providing a rich code editing experience.

2.3.1 Installing

To install DAML Studio, *install the SDK*. DAML Studio isn't currently available in the Visual Studio Marketplace.

2.3.2 Creating your first DAML file

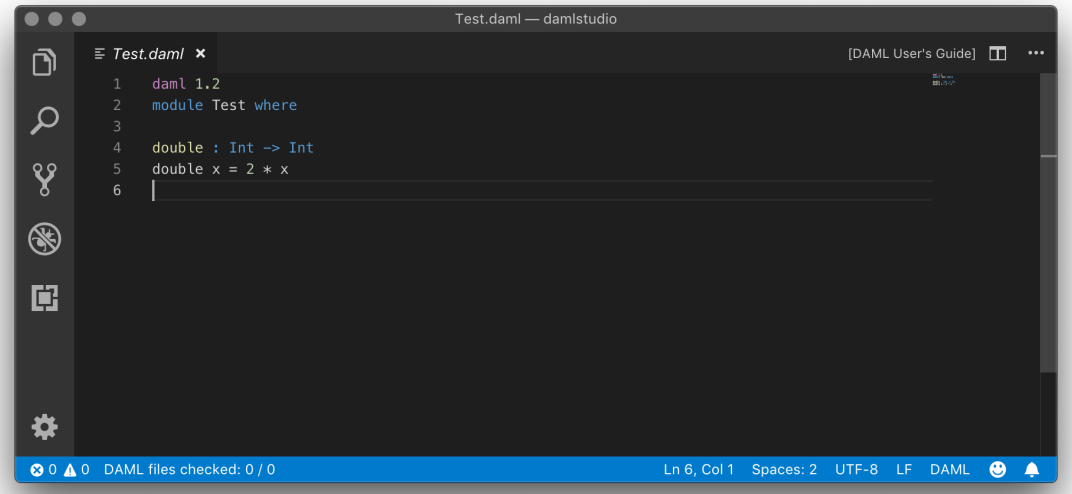
1. Start DAML Studio by running `daml studio` in the current project.
   This command starts Visual Studio Code and (if needs be) installs the DAML Studio extension, or upgrades it to the latest version.
2. Make sure the DAML Studio extension is installed:
   1. Click on the Extensions icon at the bottom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



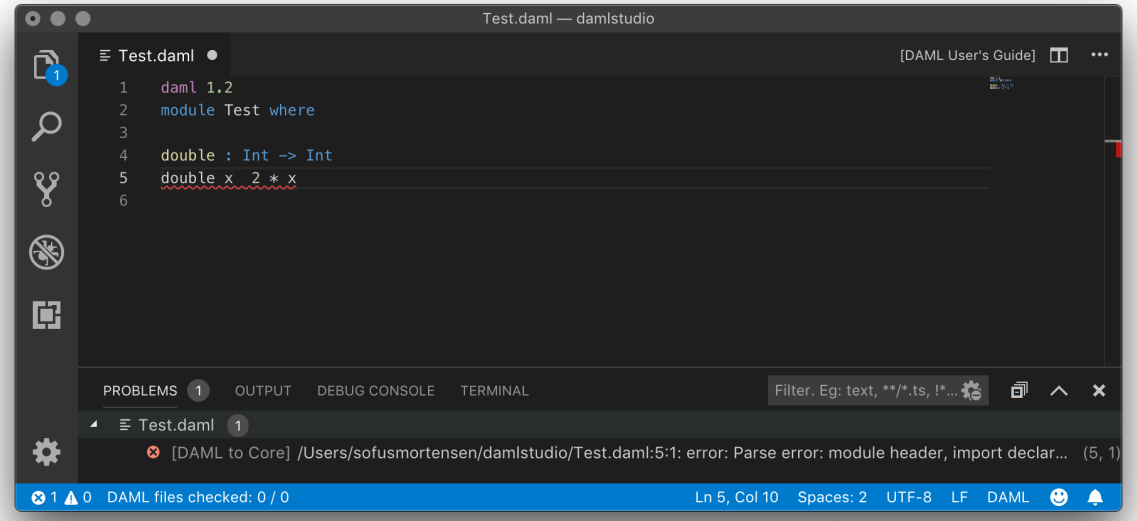
3. Open a new file (`N`) and save it (`S`) as `Test.daml`.
4. Copy the following code into your file:

```
-- Copyright (c) 2019 The DAML Authors. All rights reserved.
-- SPDX-License-Identifier: Apache-2.0

daml 1.2
module Test where

double : Int -> Int
double x = 2 * x
```

Your screen should now look like the image below.



5. Introduce a parse error by deleting the = sign and then clicking the symbol on the lower-left corner. Your screen should now look like the image below.



6. Remove the parse error by restoring the = sign.

We recommend reviewing the Visual Studio Code documentation to learn more about how to use it. To learn more about DAML, see *Language reference docs*.

2.3.3 Supported features

Visual Studio Code provides many helpful features for editing DAML files and Digital Asset recommends 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:

2.3.3.1 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

2.3.3.2 Hover tooltips

You can hover over most symbols in the code to display additional information such as its type.

2.3.3.3 Scenario results

Top-level declarations of type `Scenario` are decorated with a `Scenario results` code lens. You can click on the `Scenario results` code lens to inspect the transaction graph or an error resulting from running that scenario.

The scenario results present a simplified view of a ledger, in the form of a transaction graph, after execution of the scenario. 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:

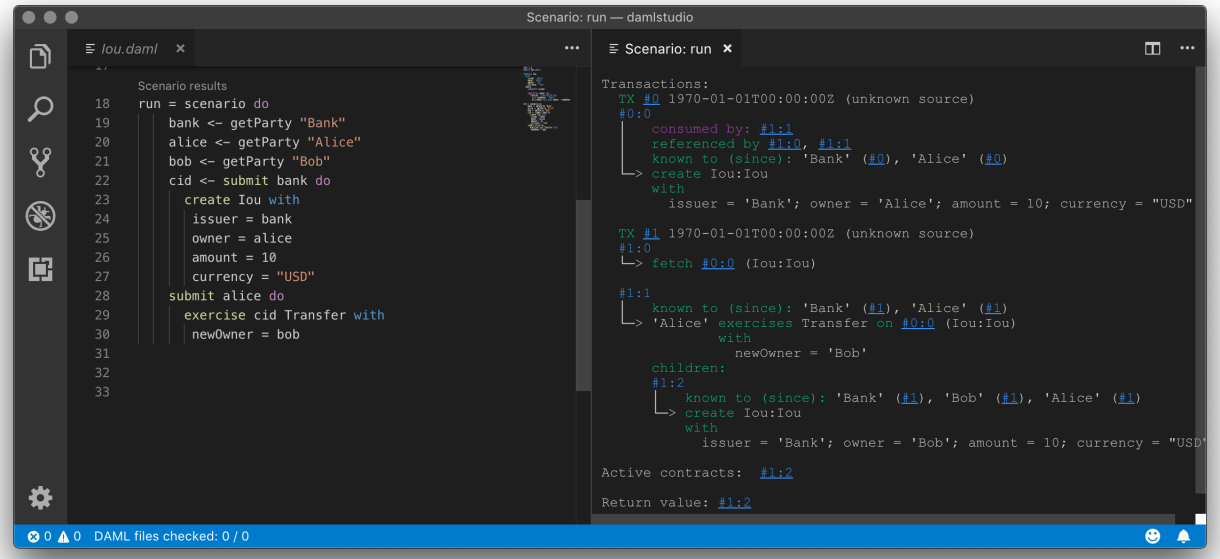


Fig. 1: Scenario results (click to zoom)

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:20:12`, which means the `Iou` module, line 20 and column 1). You can also navigate between transactions by clicking on the transaction and contract ids (e.g. `#1:0`).

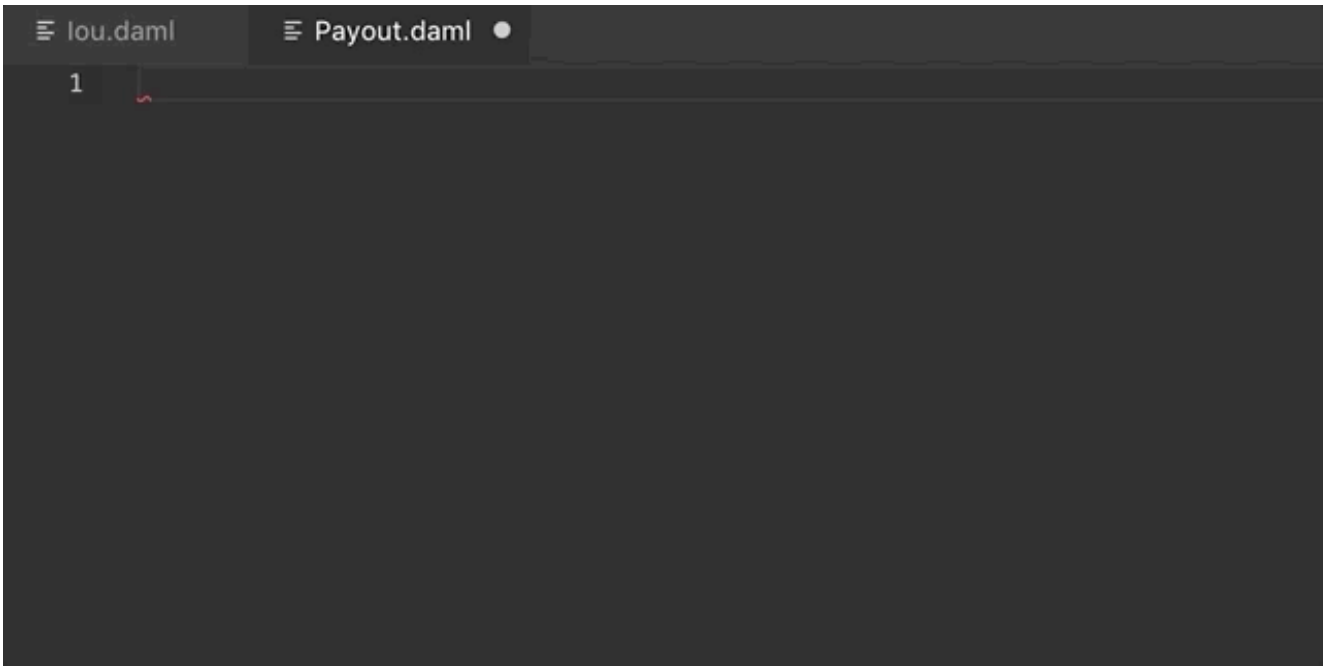
2.3.3.4 DAML snippets

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,
  "editor.quickSuggestions": false
}
```

With those changes in place, you can simply hit `Tab` after a keyword to insert the code pattern.



You can develop your own snippets by following the instructions in Creating your own Snippets to create an appropriate `daml.json` snippet file.

2.3.4 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:

Last source location A link to the last source code location encountered before the error occurred.
Environment The variables that are in scope when the error occurred. Note that contract identifiers are links that lead you to the transaction in which the contract was created.
Ledger time The ledger time at which the error occurred.
Call stack Call stack shows the function calls leading to the failing function. Updates and scenarios that do not take parameters are not included in the call stack.
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.

2.3.4.1 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!"
```

2.3.4.2 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.

2.3.4.3 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
```

(continues on next page)

(continued from previous page)

```
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.

2.3.4.4 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'
```

(continues on next page)

(continued from previous page)

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

Partial transaction:

Committed transactions:
  TX #0 1970-01-01T00:00:00Z (unknown source)
  #0:0
  │   known to (since): 'Alice' (#0)
  └─> create NotVisibleFailure:Example
      with
        owner = 'Alice'
```

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

2.4 Testing using scenarios

DAML has a built-in mechanism for testing templates called *scenarios*.

Scenarios emulate the ledger. You can 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 the sandbox ledger or ledger server. *DAML Studio* shows you the resulting Transaction graph.

For more on how scenarios work, see the *Examples* below.

On this page:

- *Scenario syntax*
  - *Scenarios*
  - *Transaction submission*
  - *Asserting transaction failure*
  - *Full syntax*
- *Running scenarios in DAML Studio*
- *Examples*
  - *Simple example*
  - *Example with two updates*
  - *Example with submitMustFail*

2.4.1 Scenario syntax

2.4.1.1 Scenarios

```
example =
  scenario do
```

A `scenario` emulates the ledger, in order to test that a DAML template or sequence of templates are working as they should.

It consists of a sequence of transactions to be submitted to the ledger (after `do`), together with success or failure assertions.

2.4.1.2 Transaction submission

```
-- Creates an instance of the Payout contract, authorized by "Alice"
submit alice do
```

The *submit* function attempts to submit a transaction to the ledger on behalf of a `Party`.

For example, a transaction could be *creating* a contract instance on the ledger, or *exercising* a choice on an existing contract.

2.4.1.3 Asserting transaction failure

```
submitMustFail alice do
  exercise payAlice Call
```

The *submitMustFail* function asserts that submitting a transaction to the ledger would fail.

This is essentially the same as `submit`, except that the scenario tests that the action doesn't work.

2.4.1.4 Full syntax

For detailed syntax, see *Reference: scenarios*.

2.4.2 Running scenarios in DAML Studio

When you load a file that includes scenarios into *DAML Studio*, it displays a Scenario results link above the scenario. Click the link to see a representation of the ledger after the scenario has run.

2.4.3 Examples

2.4.3.1 Simple example

A very simple scenario looks like this:

```
example =
  scenario do
    -- Creates the party Alice
    alice <- getParty "Alice"
    -- Creates an instance of the Payout contract, authorized by "Alice"
    submit alice do
      create Payout
        -- There's only one party: "Alice" is both the receiver and giver.
        with receiver = alice; giver = alice
```

In this example, there is only one transaction, authorized by the party `Alice` (created using `getParty "Alice"`). The ledger update is a `create`, and has to include the *arguments for the template* (`Payout with receiver = alice; giver = alice`).

2.4.3.2 Example with two updates

This example tests a contract that gives both parties an explicit opportunity to agree to their obligations.

```
example =
  scenario do
    -- Bank of England creates a contract giving Alice the option
    -- to be paid.
    bankOfEngland <- getParty "Bank of England"
    alice <- getParty "Alice"
    payAlice <- submit bankOfEngland do
      create CallablePayout with
        receiver = alice; giver = bankOfEngland

    -- Alice exercises the contract, and receives payment.
    submit alice do
      exercise payAlice Call
```

In the first transaction of the scenario, party `bankOfEngland` (created using `getParty "Bank of England"`) creates an instance of the `CallablePayout` contract with `alice` as the receiver and `bankOfEngland` as the giver.

When the contract is submitted to the ledger, it is given a unique contract identifier of type `ContractId CallablePayout`. `payAlice <-` assigns that identifier to the variable `payAlice`.

In the second statement, `exercise payAlice Call`, is an exercise of the `Call` choice on the contract instance identified by `payAlice`. This causes a `Payout` agreement with her as the `receiver` to be written to the ledger.

The workflow described by the above scenario models both parties explicitly exercising their rights and accepting their obligations:

> Party `"Bank of England"` is assumed to know the definition of the `CallablePayout` contract template and the consequences of submitting a contract instance to the ledger.
> Party `"Alice"` is assumed to know the definition of the contract template, as well as the consequences of exercising the `Call` choice on it. If `"Alice"` does not want to receive five pounds, she can simply not exercise that choice.

2.4.3.3 Example with submitMustFail

Because exercising a contract (by default) archives a contract, once party `"Alice"` exercises the `Call` choice, she will be unable to exercise it again.

To test this expectation, use the `submitMustFail` function:

```
exampleDoubleCall =
  scenario do
    bankOfEngland <- getParty "Bank of England"
    alice <- getParty "Alice"
    -- Bank of England creates a contract giving Alice the option
    -- to be paid.
    payAlice <- submit bankOfEngland do
      create CallablePayout with
        receiver = alice; giver = bankOfEngland

    -- Alice exercises the contract, and receives payment.
    submit alice do
```

(continues on next page)

(continued from previous page)

```
      exercise payAlice Call

    -- If Alice tries to exercise the contract again, it must
    -- fail.
    submitMustFail alice do
      exercise payAlice Call
```

When the `Call` choice is exercised, the contract instance is archived. The `fails` keyword checks that if `'Alice'` submits `exercise payAlice Call` again, it would fail.

2.5 Troubleshooting

Error: <X> is not authorized to commit an update
Error Argument is not of serializable type
Modelling questions
- *How to model an agreement with another party*
- *How to model rights*
- *How to void a contract*
- *How to represent off-ledger parties*
- *How to limit a choice by time*
- *How to model a mandatory action*
- *When to use Optional*

Testing questions
- *How to test that a contract is visible to a party*
- *How to test that an update action cannot be committed*

2.5.1 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.

2.5.2 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
```

(continues on next page)

(continued from previous page)

```
    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.
```

2.5.3 Modelling questions

2.5.3.1 How to model an agreement with another party

To enter into an agreement, create a contract instance 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.

2.5.3.2 How to model rights

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

2.5.3.3 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.

2.5.3.4 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.

2.5.3.5 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.

2.5.3.6 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.

2.5.3.7 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:

```
-- isJust 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.

2.5.4 Testing questions

2.5.4.1 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))
```

2.5.4.2 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.

2.6 Writing good DAML

2.6.1 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 DA/DAML application world.

`Download all the example code`

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 instance 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.

2.6.1.1 Initiate and Accept

The Initiate and Accept pattern demonstrates how to start a bilateral workflow. One party initiates by creating a proposal or an invite contract. This gives another party the chance to accept, reject or renegotiate.

Motivation

It takes two to tango, but one party has to initiate. There is no difference in business world. The contractual relationship between two businesses often starts with an invite, a business proposal, a bid offering, etc.

Invite When a market operator wants to set up a market, they need to go through an on-boarding process, in which they invite participants to sign master service agreements and fulfill different roles in the market. Receiving participants need to evaluate the rights and responsibilities of each role and respond accordingly.

Propose When issuing an asset, an issuer is making a business proposal to potential buyers. The proposal lays out what is expected from buyers, and what they can expect from the issuer. Buyers need to evaluate all aspects of the offering, e.g. price, return, and tax implications, before making a decision.

The Initiate and Accept pattern demonstrates how to write a DAML program to model the initiation of an inter-company contractual relationship. DAML modelers often have to follow this pattern to ensure no participants are forced into an obligation.

Implementation

The Initiate and Accept pattern in general involves 2 contracts:

Initiate contract The Initiate contract can be created from a role contract or any other point in the workflow. In this example, initiate contract is the proposal contract *CoinIssueProposal* the issuer created from from the master contract *CoinMaster*.

```
template CoinMaster
  with
    issuer: Party
```

(continues on next page)

(continued from previous page)

```
  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 = []
```

Trade-offs

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

2.6.1.2 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

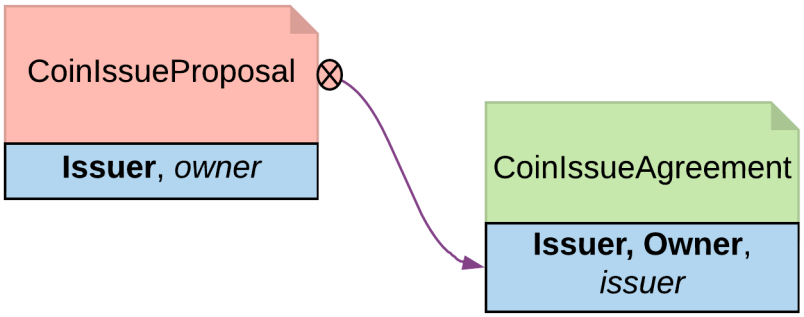


Fig. 2: Initiate and Accept pattern diagram

the ledger, filling in themselves in all the signatory fields. The Agreement contract is not created on the ledger until all parties have agreed to the Pending contract, and replaced the initiator's signature with their own.

Motivation

The *Initiate and Accept* shows how to create bilateral agreements in DAML. However, a project or a workflow often requires more than two parties to reach a consensus and put their signatures on a multi-party contract. For example, in a large construction project, there are at least three major stakeholders: Owner, Architect and Builder. All three parties need to establish agreement on key responsibilities and project success criteria before starting the construction.

If such an agreement were modeled as three separate bilateral agreements, no party could be sure if there are conflicts between their two contracts and the third contract between their partners. If the *Initiate and Accept* were used to collect three signatures on a multi-party agreement, unnecessary restrictions would be put on the order of consensus and a number of additional contract templates would be needed as the intermediate steps. Both solution are suboptimal.

Following the Multiple Party Agreement pattern, it is easy to write an agreement contract with multiple signatories and have each party accept explicitly.

Implementation

Agreement contract The *Agreement* contract represents the final agreement among a group of stakeholders. Its content can vary per business case, but in this pattern, it always has multiple signatories.

```
template ContractPlaceholder
  with
    signatories: [Party]
  where
    signatory signatories
    ensure
      unique signatories
  -- The rest of the template to be agreed to would follow here
```

Pending contract The *Pending* contract needs to contain the contents of the proposed *Agreement* contract, as a parameter. This is so that parties know what they are agreeing to, and also so that when all parties have signed, the *Agreement* contract can be created.

The *Pending* contract has a list of parties who have signed it, and a list of parties who have yet to sign it. If you add these lists together, it has to be the same set of parties as the `signatories` of the *Agreement* contract.
All of the `toSign` parties have the choice to `Sign`. This choice checks that the party is indeed a member of `toSign`, then creates a new instance of the *Pending* conract where they have been moved to the `signed` list.

```
template Pending
  with
    finalContract: ContractPlaceholder
    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 ContractPlaceholder with
        signer : Party
      controller signer
        do
          -- Check that all the required signatories have signed
↪Pending
          assert (sort alreadySigned == sort finalContract.signatories)
          create finalContract
```

Collecting the signatures in practice Since the final Pending contract has multiple signatories, **it cannot be created in that state by any one stakeholder**.
However, a party can create a pending contract, with all of the other parties in the `toSign` list.

```
  parties@[person1, person2, person3, person4] <- makePartiesFrom [
↪"Alice", "Bob", "Clare", "Dave"]
  let finalContract = ContractPlaceholder with signatories = parties
```

(continues on next page)

(continued from previous page)

```
  -- 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
```

2.6.1.3 Delegation

The Delegation pattern gives one party the right to exercise a choice on behalf of another party. The agent can control a contract instance on the ledger without the principal explicitly committing the action.

Motivation

Delegation is prevalent in the business world. In fact, the entire custodian business is based on delegation. When a company chooses a custodian bank, it is effectively giving the bank the rights to hold their securities and settle transactions on their behalf. The securities are not legally possessed by the custodian banks, but the banks should have full rights to perform actions in the client's name, such as making payments or changing investments.

The Delegation pattern enables DAML modelers to model the real-world business contractual agreements between custodian banks and their customers. Ownership and administration rights can be

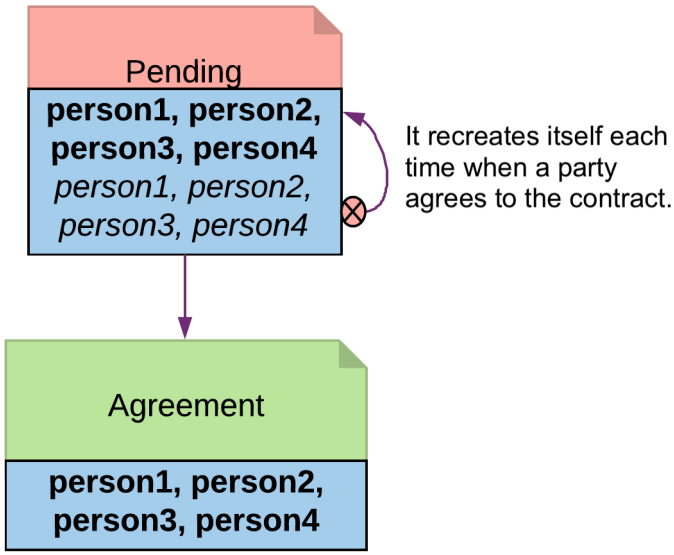


Fig. 3: Multiple Party Agreement Diagram

segregated easily and clearly.

Implementation

Pre-condition: There exists a contract, on which controller Party A has a choice and intends to delegate execution of the choice to Party B. In this example, the owner of a *Coin* contract intends to delegate the *Transfer* choice.

```
--the original contract
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
```

(continues on next page)

(continued from previous page)

```
    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
        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
```

(continues on next page)

(continued from previous page)

```
        do create this with delegates = p :: delegates
```

Note: The technique is likely to change in the future. DA is actively researching future language features for contract disclosure.

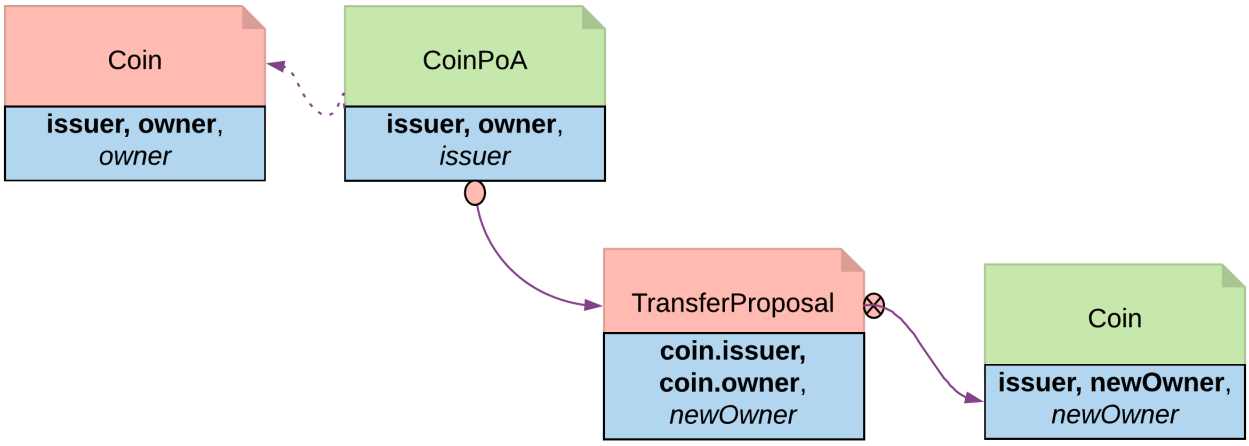


Fig. 4: Delegation pattern diagram

2.6.1.4 Authorization

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

Motivation

Authorization is an universal concept in the business world as access to most business resources is a privilege, and not given freely. For example, security trading may seem to be a plain bilateral agreement between the two trading counterparties, but this could not be further from truth. To be able to trade, the trading parties need go through a series of authorization processes and gain permission from a list of service providers such as exchanges, market data streaming services, clearing houses and security registrars etc.

The Authorization pattern shows how to model these authorization checks prior to a business transaction.

Authorization

Here is an implementation of a *Coin transfer* without any authorization:

```
template Coin
  with
    owner: Party
    issuer: Party
    amount: Decimal
```

(continues on next page)

(continued from previous page)

```
    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

    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
```

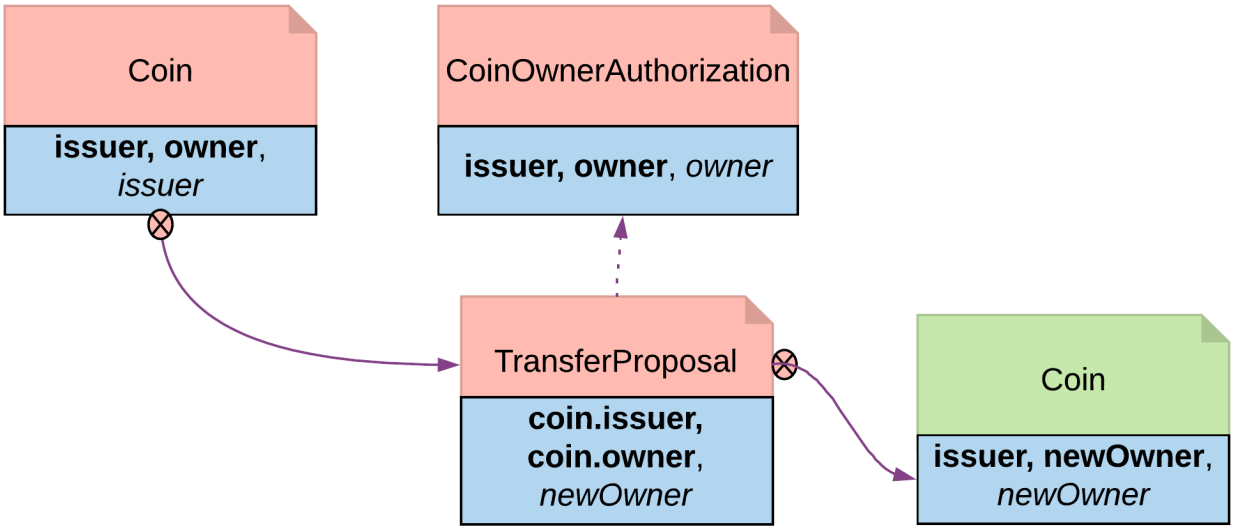


Fig. 5: Authorization Diagram

2.6.1.5 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.

Motivation

Locking is a common real-life requirement in business transactions. During the clearing and settlement process, once a trade is registered and novated to a central Clearing House, the trade is considered locked-in. This means the securities under the ownership of seller need to be locked so they cannot be used for other purposes, and so should be the funds on the buyer's account. The locked state should remain throughout the settlement Payment versus Delivery process. Once the ownership is exchanged, the lock is lifted for the new owner to have full access.

Implementation

There are three ways to achieve locking:

Locking by archiving

Pre-condition: there exists a contract that needs to be locked and unlocked. In this section, *Coin* is used as the original contract to demonstrate locking and unlocking.

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

    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
```

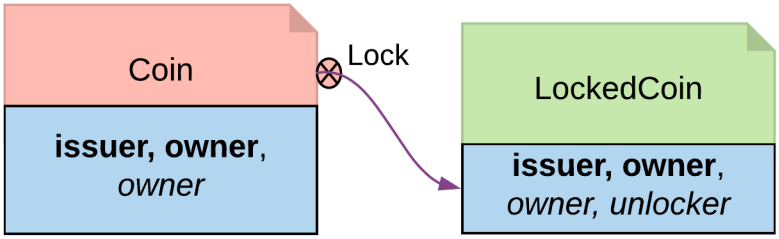


Fig. 6: Locking By Consuming Choice Diagram

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
```

(continues on next page)

(continued from previous page)

```
    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.

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
```

(continues on next page)

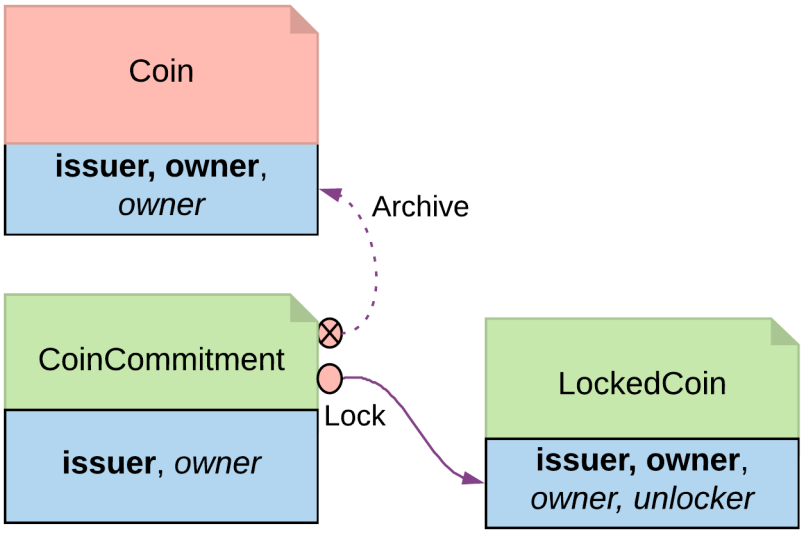


Fig. 7: Locking By Archiving Contract Diagram

(continued from previous page)

```
    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.

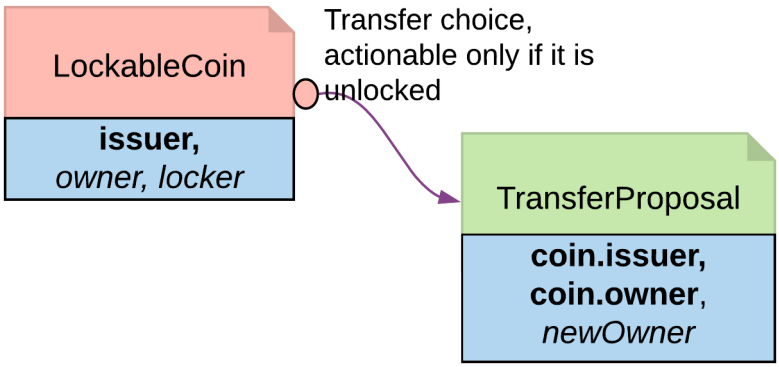
Here are the changes we made to the original *Coin* contract to make it lockable.

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
            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

    --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
```

(continues on next page)

(continued from previous page)

```
        : ()
        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

    controller locker can
      UnlockV2
        : ContractId Coin
        with coinCid : ContractId Coin
        do
          inputCoin <- fetch coinCid
```

(continues on next page)

(continued from previous page)

```
        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
```

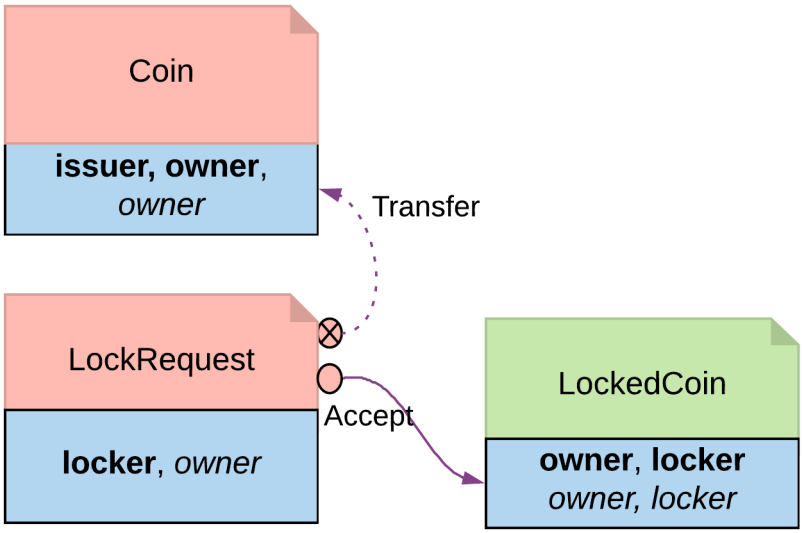


Fig. 8: Locking By Safekeeping Diagram

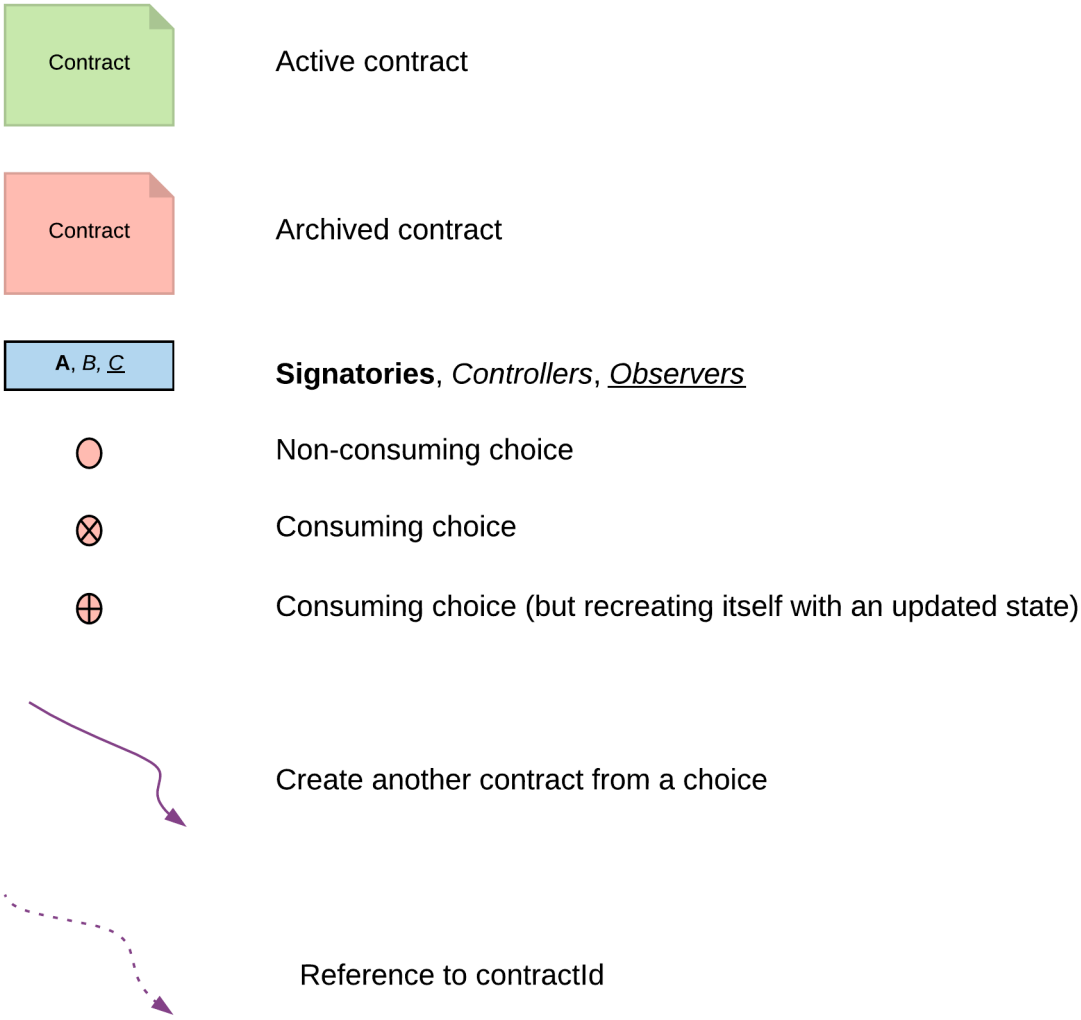
Trade-offs

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.

2.6.1.6 Diagram legends

2.6.2 Anti-patterns

This documents DLT anti-patterns, their drawbacks and more robust ways of achieving the same outcome.


Contract
Active contract
Contract
Archived contract
A, B, C
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

Don't use the ledger for orchestration
Avoid race conditions in smart contracts
Don't use status variables in smart contracts

2.6.2.1 Don't use the ledger for orchestration

Applications often need to orchestrate calculations at specific times or in a long-running sequence of steps. Examples are:

Committing assets to a settlement cycle at 10:00 am
Starting a netting calculation after trade registration has finished
Triggering the optimization of a portfolio

At first, creating a contract triggering this request might seem convenient:

```
template OptimizePortfolio
  with
    self: Party
  where
    signatory self
```

However, this is a case of using a database [ledger] for interprocess communication. This contract is a computational request from the orchestration unit to a particular program. But the ledger represents the legal rights and obligations associated with a business process: computational requests are a separate concern and shouldn't be mixed into this. Having them on-ledger has the following drawbacks:

Code bloat in shared models: introduces more things which need to be agreed upon
Limited ability to send complicated requests since they first have to be projected into smart contracts
High latency since intermediate variables have to be committed to the ledger
Changing the orchestration of a production system has a very high barrier since it may require DAML model upgrades
Orchestration contracts have no business meaning and contaminate the ledger holding business-oriented legal rights and obligations

Instead, lightweight remote procedure calls (RPC) would be more appropriate. A system designer can consider triggering the application waiting to execute a task with RPC mechanism like:

An HTTP request
A general message bus
A scheduler starting the calculation at a specific time

Notification contracts, which draw a line in the sand and have a real business meaning, don't fall under this categorization. These are persistent contracts with real meaning to the business process and not an ephemeral computational request as described above.

2.6.2.2 Avoid race conditions in smart contracts

The DLT domain lends itself to race conditions. How? Multiple parties are concurrently updating shared resources (contracts). Here's an example that's vulnerable to race conditions: a DvP where a payer allocates their asset, a receiver has to allocate their cash and then an operator does the final settlement.

```
template DvP
 with
   operator: Party
   payer: Party
   receiver: Party
   assetCid: Optional (ContractId Asset)
   cashIouCid: Optional (ContractId CashIou)
--

   controller payer can
     PayerAllocate: ContractId DvP
--

   controller receiver can
     ReceiverAllocate: ContractId DvP
--

   controller operator can
     Settle: (ContractId Asset, ContractId CashIou)
```

If the payer and receiver react to the creation of this contract and try to exercise their respective choices, one will succeed and the other will result in an attempted double-spend. Double-spends create additional work on the system because when an exception is returned, a new command needs to be subsequently generated and reprocessed. In addition, the application developer has to implement careful error handling associated with the failed command submission. It should be everyone's goal to write double-spend free code as needless exceptions dirty logs and can be a distraction when debugging other problems.

To write your code in a way that avoids race conditions, you should explicitly break up the updating of the state into a workflow of contracts which collect up information from each participant and is deterministic in execution. For the above example, deterministic execution can be achieved by refactoring the DvP into three templates:

1. `DvPRequest` created by the operator, which only has a choice for the payer to allocate.
2. `DvP` which is the result of the previous step and only has a choice for the receiver to allocate.
3. `SettlementInstruction` which is the result of the previous step. It has all the information required for settlement and can be advanced by the operator

Alternatively, if asynchronicity is required, the workflow can be broken up as follows:

1. Create a `PayerAllocation` contract to collect up the `asset`.
2. Create a `ReceiverAllocation` contract to collect up the `cashIou`.
3. Have the `Settle` choice on the `DvP` which takes the previous two contracts as arguments.

2.6.2.3 Don't use status variables in smart contracts

When orchestrating the processing of an obligation, the obligation may go through a set of states. The simplest example is locking an asset where the states are locked versus unlocked. A more complex example is the states of insurance claim:

1. Claim Requested
2. Cleared Fraud Detection
3. Approved
4. Sent for Payment

Initially, it might seem that a convenient way to represent this is with a status variable like below:

```
data ObligationStatus = ClaimRequested | ClearedFraudDetection | Approved |
↪SentForPayment deriving (Eq, Show)

template Obligation
  with
    insuranceUnderwriter: Party
    claimer: Party
    status : ObligationStatus
```

Instead, you can break up the obligation into separate contracts for each of the different states.

```
template ClaimRequest
  with
    insuranceUnderwriter: Party
    claimer: Party

template ClaimClearedFraudDetection
  with
    insuranceUnderwriter: Party
    claimer: Party
```

The drawbacks of maintaining status variables in contracts are:

- It is harder to understand the state of the ledger since you have to inspect contracts
- More complex application code is required since it has to condition on the state the contract
- Within the contract code, having many choices on a contract can make it ambiguous as to how to advance the workflow forward
- The contract code can become complex supporting all the various way to update its internal state
- Information can be leaked to parties who are not involved in the exercising of a choice
- It is harder to update the ledger/models/application if a new state is introduced
- Increased error checking code required to verify the state transitions are correct
- Makes the code harder to reason about

By breaking the contract up and removing the status variable, it eliminates the above drawbacks and makes the system transparent in its state and how to evolve forward.

2.6.3 What functionality belongs in DAML models versus application code?

The answer to this question depends on how you're using your ledger and what is important to you. Consider two different use cases: a ledger encoding legal rights and obligations between companies versus using a ledger as a conduit for internal data synchronization. Each of these solutions would be deployed in very different environments and are on either end of the trust and coordination spectrums. Internally to a company, trust is high and the ability to coordinate change is high. External to a company, the opposite is true.

The rest of this page will talk about how to organize things in either case. For your particular solution, it is important to similarly identify the what factors are important to you, then separate along those lines.

Looking at the ledger from a legal perspective
Looking at the ledger from a data synchronization perspective

2.6.3.1 Looking at the ledger from a legal perspective

When the ledger is encoding legal rights and obligations between external counterparties, a defensive/minimalistic approach to functionality in DAML models may be prudent. The reasons for this are:

- It is a litigious environment where the ledger's state may require examination in court
- The ledger is a valuable source of legal information and shouldn't be contaminated with non-business oriented logic
- The more functionality in shared models, the more which needs to be agreed upon upfront by all companies involved. Further updating shared models is hard since all companies need to coordinate

As a result, shared functionality in DAML models needs careful scrutinization. This minimalistic approach might only include:

- Contracts representing, and going into the servicing of, traditional legal contracts
- Contracts narrowly associated with the business process such as obligations for payment/delivery
- Contractual eligibility checks prior to obligation creation - e.g. prerequisites for creating an insurance claim
- Operations requiring atomicity such as swapping of ownership
- Calculations resulting in legal obligations such as the payout of a call option

Functionality not going into the DAML models then must go into the application. These non-business oriented items may include:

- Commonly available libraries like calendars or date calculations
- Code to parse messages - e.g. FIX trade confirmation messages
- Code to orchestrate a batch calculation
- Calculations specific to a participant

2.6.3.2 Looking at the ledger from a data synchronization perspective

On the other hand, when doing data synchronization most of the inter-process communication between parties belongs on the ledger. This perspective is grounded in the fact that DA's platform acts as a messaging bus where the messages are subject to certain guarantees:

- The initiating party is authentic
- Messages conform to DAML model specification
- Messages are approved by all participants hosting stakeholders of the message

Therefore, when doing data synchronization all of the above functionality is eligible to go into the DAML models and have the application be a lightweight router. However, there are still some things for which it isn't sensible to put on the ledger. For examples of these, see the section on *Anti-patterns*.

Chapter 3

Building applications

3.1 Writing applications using the Ledger API

3.1.1 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*.

3.1.1.1 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`.

3.1.1.2 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 instance, 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.

Commands can be labeled with two application-specific IDs, both of which are returned in completion events:

A *commandId*, returned to the submitting application only. It is generally used to implement this correlation between commands and 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 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*.

3.1.1.3 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 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 currently active on the ledger.

The active contracts service returns the current contract set as a set of created events that would re-create the state being reported. Each created event has a ledger offset where it occurs. You can infer the ledger offset of the contract set from the ledger offset of the last event you receive.

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.

3.1.1.4 Utility services

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 maximum and minimum values for the difference in Ledger Effective Time and Maximum Record Time (see *Time Service* for details of these).

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

3.1.1.5 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.

This is important because you have to include two timestamps when you submit a command - the *Ledger Effective Time (LET)*, and the *Maximum Record Time (MRT)*. For the command to be accepted, LET must be greater than the current ledger time.

MRT is used in the detection of lost commands.

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

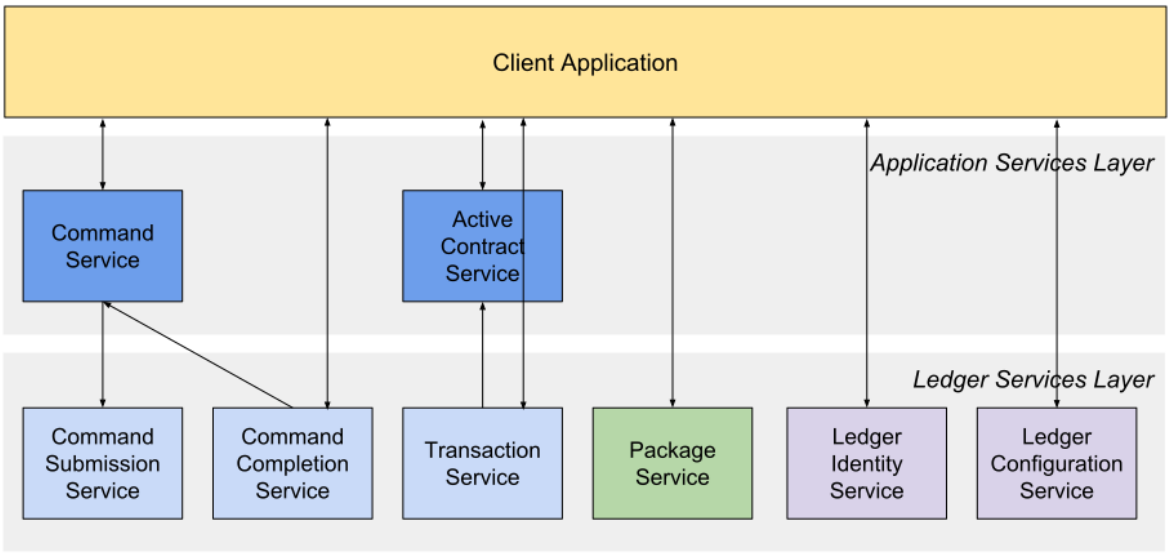
Reset service

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*.

3.1.1.6 Services diagram



3.1.2 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*.

3.1.2.1 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`.

3.1.2.2 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`.

3.1.2.3 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 }`

3.1.2.4 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 }
```

3.1.2.5 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
↪}
```

Similarly, the generic template definition

```
template Template a => Proposal a
  with
    proposal: a
    proposers: [Party]
    receivers: [Party]
  where
```

results in the record declaration:

```
data Proposal a = Proposal { proposal: a; proposers: [Party]; receivers:
↪[Party] }
```

which translates to the DAML-LF record declaration:

```
record Proposal a  { proposal: a; proposers: List Party; receivers: List
↪Party }
```

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 }
```

Generic template instantiations

In the context of the templates `Iou` and `Proposal` above, the generic template instantiation

```
template instance IouProposal = Proposal Iou
```

does not produce a DAML data definition but only the DAML-LF record declaration:

```
record IouProposal  { proposal: Iou; proposers: List Party; receivers: List
↪Party }
```

Note that `IouProposal` is a copy of `Proposal` with the type variable `a` replaced by `Iou` on the right hand side.

In DAML, `IouProposal` can also be used as a type synonym for `Proposal Iou`.

DAML contracts are stored on a ledger. In order to exercise choices on those contracts, create new ones, or read from the ledger, you need to use the **Ledger API**. (Every ledger that DAML can run on exposes this same API.) To write an application around a DAML ledger, you'll need to interact with the Ledger API from another language.

3.1.3 Resources available to you

The Java bindings: a library to help you write idiomatic applications using the Ledger API in Java.
Read the documentation for the Java bindings
The experimental Node.js bindings: a library to help you write idiomatic applications using the Ledger API in JavaScript. Information about the Node.js bindings isn't available in this documentation, but is on GitHub.
Read the documentation for the Node.js bindings
The underlying gRPC API: if you want to interact with the ledger API from other languages, you'll need to use gRPC directly.
Read the documentation for the gRPC API
The application architecture guide: this documentation gives high-level guidance on designing DAML Ledger applications.
Read the application architecture guide

3.1.4 What's in the Ledger API

No matter how you're accessing it (Java bindings, Node.js bindings, or gRPC), 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 *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 minimum and maximum TTL for commands.

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.

3.1.5 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 inside the DAML SDK, 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)

3.1.5.1 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 *code generation*, 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.

3.2 Java bindings

3.2.1 Generate Java code from DAML

3.2.1.1 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.

3.2.1.2 Download

You can download the latest version of the Java codegen. Make sure that the following versions are aligned:

the downloaded Java codegen jar file, eg. 10x.y.z
the dependency to *bindings-java*, eg. 10x.y.z
the `sdk-version` attribute in the *daml.yaml* file, eg. x.y.z

3.2.1.3 Run the Java codegen

The Java codegen takes DAML archive (DAR) files as input and generates Java files for DAML templates, records, and variants. For information on creating DAR files see *Building DAML projects*. To use the Java codegen, run this command in a terminal:

```
java -jar <path-to-codegen-jar>
```

Use this command to display the help text:

```
java -jar codegen.jar --help
```

Generate Java code from DAR files

Pass one or more DAR files as arguments to the Java codegen. Use the `-o` or `--output-directory` parameter for specifying the directory for the generated Java files.

```
java -jar java-codegen.jar -o target/generated-sources/daml daml/my-
↪project.dar
                            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

To avoid possible name clashes in the generated Java sources, you should specify a Java package prefix for each input file:

```
java -jar java-codegen.jar -o target/generated-sources/daml \
    daml/project1.dar=com.example.daml.project1 \
                      ^^^^^^^^^^^^^^^^^^^^^^^^^^
    daml/project2.dar=com.example.daml.project2
                      ^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Generate the decoder utility class

When reading transactions from the ledger, you typically want to convert a CreatedEvent from the Ledger API to the corresponding generated `Contract` class. The Java codegen can optionally generate a decoder class based on the input DAR files that calls the `fromCreatedEvent` method of the respective generated `Contract` class (see *Templates*). The decoder class can do this for all templates in the input DAR files.

To generate such a decoder class, provide the command line parameter `-d` or `--decoderClass` with a fully qualified class name:

```
java -jar java-codegen.jar -o target/generated-sources/daml \
    -d com.myproject.DamModelDecoder daml/my-project.dar
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Receive feedback

By default, the logging is configured so that you'll only see error messages.

If you want to change this behavior, you can ask to receive more extensive feedback using the `-V` or `--verbosity` command-line option. This option takes a numeric parameter from 0 to 4, where 0 corresponds to the default quiet behavior and 4 represents the most verbose output possible.

In the following example the logging is set to print most of the output with detailed debugging information:

```
java -jar java-codegen.jar -o target/generated-sources/daml -V 3
                                                            ^^^^
```

Integrate with build tools

While we currently don't provide direct integration with Maven, Groovy, SBT, etc., you can run the Java codegen as described in *Run the Java codegen* just like any other external process (for example the protobuf compiler). Alternatively you can integrate it as a runnable dependency in your `pom.xml` file for Maven.

The following snippet is an excerpt from the `pom.xml` that is part of the *Quickstart guide* guide.

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
    <version>1.6.0</version>
    <dependencies>
        <dependency>
            <groupId>com.daml.java</groupId>
            <artifactId>codegen</artifactId>
            <version>__VERSION__</version>
            <type>jar</type>
        </dependency>
    </dependencies>
    <executions>
        <execution>
            <id>daml-codegen-java</id>
```

(continues on next page)

(continued from previous page)

```
            <phase>generate-sources</phase>
            <goals>
                <goal>java</goal>
            </goals>
            <configuration>
                <includeProjectDependencies>false</
↪includeProjectDependencies>
                <includePluginDependencies>true</
↪includePluginDependencies>
                <mainClass>com.digitalasset.daml.lf.codegen.Main</
↪mainClass>
                <arguments>
                    <argument>-o</argument>
                    <argument>${daml-codegen-java.output}</
↪argument>
                    <argument>-d</argument>
                    <argument>com.digitalasset.quickstart.iou.
↪TemplateDecoder</argument>
                    <argument>${project.basedir}/.daml/dist/
↪quickstart-0.0.1.dar=com.digitalasset.quickstart.model</argument>
                </arguments>
            </configuration>
        </execution>
        <execution>
</plugin>
<plugin>
```

3.2.1.4 Compile the generated Java code

To compile the generated Java code, add the *Java Bindings* library with the same version as the Java codegen to the classpath.

With Maven you can do this by adding a `dependency` to the `pom.xml` file:

```
<dependency>
    <groupId>com.daml.ledger</groupId>
    <artifactId>bindings-rxjava</artifactId>
    <version>x.y.z</version>
</dependency>
```

3.2.1.5 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
`Decimal`	`java.math.BigDecimal`	Decimal
`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.	TextMap
`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 1: DAML

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

Listing 2: 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 3: Com/Acme.daml

```
daml 1.2
module Com.Acme 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 4: com/acme/Person.java

```
package com.acme;

public class Person {
  public final Name name;
  public final BigDecimal age;

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

  public Person(Name name, BigDecimal age) { /* ... */ }
  public Record toValue() { /* ... */ }
}
```

A Java file is generated that defines the class for the type `Name`:

Listing 5: com/acme/Name.java

```
package com.acme;

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

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

  public Name(String fistName, String lastName) { /* ... */ }
  public Record 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 6: Com/Acme.daml

```
daml 1.2
module Com.Acme 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

    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 7: com/acme/Bar.java

```
package com.acme;
```

(continues on next page)

(continued from previous page)

```
public class Bar extends Template {

  public static final Identifier TEMPLATE_ID = new Identifier("some-
↪package-id", "Com.Acme", "Bar");

  public final String owner;
  public final String name;

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

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

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

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

  public static class ContractId {
    public final String contractId;

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

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

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

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

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

Note that the static methods returning an `ExerciseByKeyCommand` will only be generated for templates that define a key.

Variants (a.k.a sum types)

A *variant or sum type* is a type with multiple constructors, where each constructor wraps a value of another type. The generated code is comprised of an abstract class for the variant type itself and a subclass thereof for each constructor. Classes for variant constructors are similar to classes for records.

Listing 8: Com/Acme.daml

```
daml 1.2
module Com.Acme where

data BookAttribute = Pages Int
                   | Authors [Text]
                   | Title Text
                   | Published with year: Int; publisher: Text
```

The Java code generated for this variant is:

Listing 9: com/acme/BookAttribute.java

```
package com.acme;

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

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

Listing 10: com/acme/bookattribute/Pages.java

```
package com.acme.bookattribute;

public class Pages extends BookAttribute {
  public final Long longValue;

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

  public Pages(Long longValue) { /* ... */ }
  public Value toValue() { /* ... */ }
}
```

Listing 11: com/acme/bookattribute/Authors.java

```
package com.acme.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 12: com/acme/bookattribute/Title.java

```
package com.acme.bookattribute;

public class Title extends BookAttribute {
  public final String stringValue;

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

  public Title(String stringValue) { /* ... */ }
  public Value toValue() { /* ... */ }
}
```

Listing 13: com/acme/bookattribute/Published.java

```
package com.acme.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 Record 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 14: Com/Acme.daml

```
daml 1.2
module Com.Acme where

data Attribute a = Attribute
    with v : a

data BookAttributes = BookAttributes with
   pages : (Attribute Int)
   authors : (Attribute [Text])
```

(continues on next page)

(continued from previous page)

```
  title : (Attribute Text)
```

The Java codegen generates a Java file with a generic class for the `Attribute a` data type:

Listing 15: com/acme/Attribute.java

```
package com.acme;

public class Attribute<a> {
  public final a value;

  public Attribute(a value) { /* ... */ }

  public Record 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 16: Com/Acme.daml

```
daml 1.2
module Com.Acme where

data Color = Red | Blue | Green
```

The Java code generated for this variant is:

Listing 17: com/acme/Color.java

```
package com.acme;


public enum Color {
  RED,

  GREEN,

  BLUE;

  /* ... */

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

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

Listing 18: com/acme/bookattribute/Authors.java

```
package com.acme.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 List and Optional values

The conversion of the Java `List` and `Optional` types requires multiple steps to first convert the list elements or the value inside the optional creating the DamlList or DamlOptional value.

```
Attribute<List<String>> authorsAttribute = new Attribute<List<String>>
↪(Arrays.asList("Homer", "Ovid", "Vergil"));

Value serializedAuthors = authorsAttribute.toValue(f -> new DamlList(f.
↪stream().map(Text::new).collect(Collectors.<Value>toList())));
```

The conversion of the Java `List` and `Optional` types similarly require that the Java Bindings types DamlList and DamlOptional are converted to it's Java equivalent and then all the contained elements are converted to Java types.

```
Attribute<List<String>> authorsAttribute = Attribute.<List<String>>
↪fromValue(serializedAuthors,
    f0 -> f0.asList().orElseThrow(() -> new IllegalArgumentException(
↪"Expected DamlList field")).getValues().stream()
        .map(f1 -> f1.asText().orElseThrow(() -> new
↪IllegalArgumentException("Expected Text element")).getValue())
            .collect(Collectors.toList()));
```

3.2.2 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.

3.2.2.1 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.

3.2.2.2 Example project – Ping Pong without reactive components

PingPongMain.java

The entry point for the Java code is the main class `src/main/java/examples/pingpong/PingPongMain.java`. Look at this class to see:

how to connect to and interact with the 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.

3.2.2.3 Example project – Ping Pong with reactive components

PingPongMain.java

The entry point for the Java code is the main class `src/main/java/examples/pingpong/PingPongMain.java`. Look at this class to see:

how to connect to and interact with the DAML Ledger via the Java bindings
how to use the Reactive Components to build an automation for both parties

PingPongBot

At high level, this application follows the same steps as the one without Reactive Components except for the `PingPongProcessor`. In this application, the `PingPongProcessor` is replaced by the `PingPongBot`.

The `PingPongBot` has two important methods:

> `getContractInfo(Record record, TransactionContext context)` which is used to get the information useful to the application from a created contract and the context
> `process(LedgerView<ContractInfo> ledgerView)` which implements the logic of the application by converting the local view of the Ledger into a stream of `Commands`

Output

The application prints statements similar to the ones seen in the section above.

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.

3.2.3 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 the 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.
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`.

3.2.3.1 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*.

3.2.3.2 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 DA 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.

3.2.3.3 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.

3.2.3.4 Writing automations: Bot

The `Bot` is an abstraction used to write automation for the 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,
```

(continues on next page)

(continued from previous page)

```
        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`.

3.2.4 Reference documentation

Click here for the JavaDoc reference documentation.

3.2.5 Getting started

The Java bindings library can be added to a Maven project.

3.2.5.1 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.

Note: As of DAML SDK release 0.13.3, the Java Bindings libraries are available via the public Maven Central repository. Earlier releases are available from the DAML Bintray repository.

You can also take a look at the `pom.xml` file from the *quickstart project*.

3.2.5.2 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.

3.2.5.3 Connecting securely

The Java bindings library lets you connect to a DAML Ledger via a secure connection. The factory methods `DamlLedgerClient.forLedgerIdAndHost` and `DamlLedgerClient.forHostWithLedgerIdDiscovery` accept a parameter of type `Optional<SslContext>`. If the value of that optional parameter is not present (i.e. by passing `Optional.empty()`), a plain text / insecure connection will be established. This is useful when connecting to a locally running Sandbox.

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.

3.2.5.4 Advanced connection settings

Sometimes the default settings for gRPC connections/channels are not suitable for a given situation. These usecases are supported by creating a a custom ManagedChannel object via ManagedChannelBuilder or NettyChannelBuilder and passing the channel instance to the constructor of DamlLedgerClient.

3.2.6 Example project

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

3.3 Scala bindings

This page provides a basic Scala programmer's introduction to working with Digital Asset distributed ledger, using the Scala programming language and the **Ledger API**.

3.3.1 Introduction

The Scala bindings is a client implementation of the **Ledger API**. The Scala bindings library lets you write applications that connect to the Digital Asset distributed 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

3.3.2 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.scala" %% "bindings" % daSdkVersion,
)

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

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`)
```

3.3.3 Generating Scala code

1) Install *the latest version of the DAML 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.digitalasset.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).

3.3.4 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.

3.3.4.1 Create a contract and send a CreateCommand

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

```
import com.digitalasset.ledger.client.binding.{Primitive => P}
import com.digitalasset.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.digitalasset.quickstart.iou.ClientUtil#submitCommand.

3.3.4.2 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.digitalasset.quickstart.iou.IouMain#newOwnerAcceptsAllTransfers.

3.4 Node.js bindings

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.

3.5 The Ledger API using gRPC

3.5.1 Ledger API Reference

3.5.1.1 com/digitalasset/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
trace_context	*TraceContext*		Server side tracing will be registered as a child of the submitted context. This field is a future extension point and is currently not supported. 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
trace_context	*TraceContext*		Zipkin trace context. This field is a future extension point and is currently not supported. 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.

3.5.1.2 com/digitalasset/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

UploadDarFileResponse

An empty message that is received when the upload operation succeeded.

PackageManagementService

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. When the participant is run in mode requiring authentication, all the calls in this interface will respond with UNAUTHENTICATED, if the caller fails to provide a valid access token, and will respond with PERMISSION_DENIED, if the claims in the token are insufficient to perform a given operation. Subsequently, only specific errors of individual calls not related to authorization will be described.

Method name	Request type	Response type	Description
ListKnownPackages	*ListKnownPackagesRequest*	*ListKnownPackagesResponse*	Returns the details of all DAML-LF packages known to the backing participant. This request will always succeed.
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 UNIMPLEMENTED, if DAR package uploading is not supported by the backing participant. - Respond with INVALID_ARGUMENT, if the DAR file is too big or malformed. The maximum supported size is implementation specific.

3.5.1.3 com/digitalasset/ledger/api/v1/admin/party_management_service.proto

AllocatePartyRequest

Field	Type	Label	Description
party_id_hint	*string*		A hint to the backing participant what party id to allocate. It can be ignored. Must be a valid PartyIdString (as describe 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`).

ListKnownPartiesRequest

ListKnownPartiesResponse

Field	Type	Label	Description
party_details	*PartyDetails*	repeated	The details of all DAML parties hosted 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

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. When the participant is run in mode requiring authentication, all the calls in this interface will respond with UNAUTHENTICATED, if the caller fails to provide a valid access token, and will respond with PERMISSION_DENIED, if the claims in the token are insufficient to perform a given operation. Subsequently, only specific errors of individual calls not related to authorization will be described.

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. This method is expected to succeed provided the backing participant is healthy, otherwise it responds with INTERNAL grpc error. 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
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. This request will always succeed.
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 UNIMPLEMENTED if synchronous party allocation is not supported by the backing participant. - Respond with 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 the surface protocol, the arguments take implicit values

3.5.1.4 com/digitalasset/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. Required Must be a valid LedgerString (as described in `value.proto`).
trace_context	*TraceContext*		Server side tracing will be registered as a child of the submitted context. This field is a future extension point and is currently not supported. Optional

CompletionEndResponse

Field	Type	Label	Description
offset	*LedgerOffset*		This offset can be used in a CompletionStreamRequest message. Required

CompletionStreamRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger id reported by the Ledger Identification Service. Must be a valid LedgerString (as described in `value.proto`). 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. 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. 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 Transac-

tion Service. Commands may fail in 4 distinct manners:

1. `INVALID_PARAMETER` gRPC error on malformed payloads and missing required fields.
2. Failure communicated in the gRPC error.
3. Failure communicated in a Completion.
4. A Checkpoint with `record_time` > command `mrt` arrives through the Completion Stream, and the command's Completion was not visible before. In this case the command is lost.

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.

Interprocess tracing of command submissions may be achieved via Zipkin by filling out the `trace_context` field. The server will return a child context of the submitted one, (or a new one if the context was missing) on both the Completion and Transaction streams.

Method name	Request type	Response type	Description
CompletionStream	*CompletionStreamRequest*	*CompletionStreamResponse*	Subscribe to command completion events.
CompletionEnd	*CompletionEndRequest*	*CompletionEndResponse*	Returns the offset after the latest completion.

3.5.1.5 com/digitalasset/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
trace_context	*TraceContext*		Server side tracing will be registered as a child of the submitted context. This field is a future extension point and is currently not supported. Optional

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.

Method name	Request type	Response type	Description
SubmitAndWait	*SubmitAndWaitRequest*	.google.protobuf.Empty	Submits a single composite command and waits for its result. Returns `RESOURCE_EXHAUSTED` if the number of in-flight commands reached the maximum (if a limit is configured). Propagates the gRPC error of failed submissions including DAML interpretation errors.
SubmitAndWaitForTransactionId	*SubmitAndWaitRequest*	*SubmitAndWaitForTransactionIdResponse*	Submits a single composite command, waits for its result, and returns the transaction id. Returns `RESOURCE_EXHAUSTED` if the number of in-flight commands reached the maximum (if a limit is configured). Propagates the gRPC error of failed submissions including DAML interpretation errors.
SubmitAndWaitForTransaction	*SubmitAndWaitRequest*	*SubmitAndWaitForTransactionResponse*	Submits a single composite command, waits for its result, and returns the transaction. Returns `RESOURCE_EXHAUSTED` if the number of in-flight commands reached the maximum (if a limit is configured). Propagates the gRPC error of failed submissions including DAML interpretation errors.
SubmitAndWaitForTransactionTree	*SubmitAndWaitRequest*	*SubmitAndWaitForTransactionTreeResponse*	Submits a single composite command, waits for its result, and returns the transaction tree. Returns `RESOURCE_EXHAUSTED` if the number of in-flight commands reached the maximum (if a limit is configured). Propagates the gRPC error of failed submissions including DAML interpretation errors.

3.5.1.6 com/digitalasset/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
trace_context	*TraceContext*		Server side tracing will be registered as a child of the submitted context. This field is a future extension point and is currently not supported. Optional

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 4 distinct manners:

1) `INVALID_PARAMETER` gRPC error on malformed payloads and missing required fields.
2) Failure communicated in the gRPC error.
3) Failure communicated in a Completion.
4) A Checkpoint with `record_time` > command `mrt` arrives through the Completion Stream, and the command's Completion was not visible before. In this case the command is lost.

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.

Interprocess tracing of command submissions may be achieved via Zipkin by filling out the `trace_context` field. The server will return a child context of the submitted one, (or a new one if the context was missing) on both the Completion and Transaction streams.

Method name	Request type	Response type	Description
Submit	*SubmitRequest*	.google.protobuf.Empty	Submit a single composite command.

3.5.1.7 com/digitalasset/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 identified the command. This identifier should be unique for each new command within an application domain, i.e., the triple (application_id, party, command_id) must be unique. It can be used for matching the requests with their respective completions. Must be a valid LedgerString (as described in `value.proto`). Required
party	*string*		Party on whose behalf the command should be executed. It is up to the server to verify that the authorisation can be granted and that the connection has been authenticated for that party. Must be a valid PartyIdString (as described in `value.proto`). Required
ledger_effective_time	google.protobuf.Timestamp		MUST be an approximation of the wall clock time on the ledger server. Required
maximum_record_time	google.protobuf.Timestamp		The deadline for observing this command in the completion stream before it can be considered to have timed out. Required
commands	*Command*	repeated	Individual elements of this atomic command. Must be non-empty. Required

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

3.5.1.8 com/digitalasset/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
trace_context	*TraceContext*		The trace context submitted with the command. This field is a future extension point and is currently not supported. Optional

3.5.1.9 com/digitalasset/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 descibed 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. For *CreatedEvent'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. 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. 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

3.5.1.10 com/digitalasset/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
trace_context	*TraceContext*		Server side tracing will be registered as a child of the submitted context. This field is a future extension point and is currently not supported. Optional

GetLedgerConfigurationResponse

Field	Type	Label	Description
ledger_configuration	*LedgerConfiguration*		The latest ledger configuration.

LedgerConfiguration

LedgerConfiguration contains parameters of the ledger instance that may be useful to clients.

Field	Type	Label	Description
min_ttl	google.protobuf.Duration		Minimum difference between ledger effective time and maximum record time in submitted commands.
max_ttl	google.protobuf.Duration		Maximum difference between ledger effective time and maximum record time in submitted commands.

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.

3.5.1.11 com/digitalasset/ledger/api/v1/ledger_identity_service.proto

GetLedgerIdentityRequest

Field	Type	Label	Description
trace_context	*TraceContext*		Server side tracing will be registered as a child of the submitted context. This field is a future extension point and is currently not supported. Optional

GetLedgerIdentityResponse

Field	Type	Label	Description
ledger_id	*string*		The ID of the ledger exposed by the server. Requests submitted with the wrong ledger ID will result in `NOT_FOUND` gRPC errors. 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.

3.5.1.12 com/digitalasset/ledger/api/v1/ledger_offset.proto

LedgerOffset

Describes a specific point on the ledger.

Field	Type	Label	Description
absolute	*string*		Absolute values are acquired by reading the transactions in the stream. The offsets can be compared. The format may vary between implementations. It is either a string representing an ever-increasing integer, or a composite string containing `<block-hash>-<block-height>-<event-id>`; ordering requires comparing numerical values of the second, then the third element.
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.

3.5.1.13 com/digitalasset/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
trace_context	*TraceContext*		Server side tracing will be registered as a child of the submitted context. This field is a future extension point and is currently not supported. Optional

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
trace_context	*TraceContext*		Server side tracing will be registered as a child of the submitted context. This field is a future extension point and is currently not supported. Optional

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
trace_context	*TraceContext*		Server side tracing will be registered as a child of the submitted context. This field is a future extension point and is currently not supported. Optional

ListPackagesResponse

Field	Type	Label	Description
package_ids	*string*	repeated	The IDs of all DAML-LF packages supported by the server. Each element must be a valid PackageIdString (as described in `value.proto`). Required

HashFunction

Name	Number	Description
SHA256	0	

PackageStatus

Name	Number	Description
UNKNOWN	0	The server is not aware of such a package.
REGISTERED	1	The server is able to execute DAML commands operating on this package.

PackageService

Allows clients to query the DAML-LF packages that are supported by the server.

Method name	Request type	Response type	Description
ListPackages	*ListPackagesRequest*	*ListPackagesResponse*	Returns the identifiers of all supported packages.
GetPackage	*GetPackageRequest*	*GetPackageResponse*	Returns the contents of a single package, or a `NOT_FOUND` error if the requested package is unknown.
GetPackageStatus	*GetPackageStatusRequest*	*GetPackageStatusResponse*	Returns the status of a single package.

3.5.1.14 com/digitalasset/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.

3.5.1.15 com/digitalasset/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. On mismatch, an `INVALID_PARAMETER` gRPC error will be returned.
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.

3.5.1.16 com/digitalasset/ledger/api/v1/trace_context.proto

TraceContext

Data structure to propagate Zipkin trace information. See https://github.com/openzipkin/b3-propagation Trace identifiers are 64 or 128-bit, but all span identifiers within a trace are 64-bit. All identifiers are opaque.

Field	Type	Label	Description
trace_id_high	*uint64*		If present, this is the high 64 bits of the 128-bit identifier. Otherwise the trace ID is 64 bits long.
trace_id	*uint64*		The TraceId is 64 or 128-bit in length and indicates the overall ID of the trace. Every span in a trace shares this ID.
span_id	*uint64*		The SpanId is 64-bit in length and indicates the position of the current operation in the trace tree. The value should not be interpreted: it may or may not be derived from the value of the TraceId.
parent_span_id	google.protobuf.UInt64Value		The ParentSpanId is 64-bit in length and indicates the position of the parent operation in the trace tree. When the span is the root of the trace tree, the ParentSpanId is absent.
sampled	*bool*		When the sampled decision is accept, report this span to the tracing system. When it is reject, do not. When B3 attributes are sent without a sampled decision, the receiver should make one. Once the sampling decision is made, the same value should be consistently sent downstream.

3.5.1.17 com/digitalasset/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
trace_context	*TraceContext*		Zipkin trace context. This field is a future extension point and is currently not supported. Optional

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 triggerd this transaction. Required
trace_context	*TraceContext*		Zipkin trace context. This field is a future extension point and is currently not supported. Optional

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*		

3.5.1.18 com/digitalasset/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 data will be returned.

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*		

3.5.1.19 com/digitalasset/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
trace_context	*TraceContext*		Server side tracing will be registered as a child of the submitted context. This field is a future extension point and is currently not supported. Optional

GetLedgerEndResponse

Field	Type	Label	Description
offset	*LedgerOffset*		The absolute offset of the current ledger end.

GetTransactionByEventIdRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as described in `value.proto`). 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
trace_context	*TraceContext*		Server side tracing will be registered as a child of the submitted context. This field is a future extension point and is currently not supported. Optional

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
trace_context	*TraceContext*		Server side tracing will be registered as a child of the submitted context. This field is a future extension point and is currently not supported. Optional

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. Required
end	*LedgerOffset*		End of the requested ledger section. Optional, if not set, the stream will not terminate.
filter	*TransactionFilter*		Requesting parties with template filters. 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
trace_context	*TraceContext*		Server side tracing will be registered as a child of the submitted context. This field is a future extension point and is currently not supported. 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.
GetTransactionTrees	*GetTransactionsRequest*	*GetTransactionTreesResponse*	Read the ledger's complete transaction tree stream for a set of parties.
GetTransactionByEventId	*GetTransactionByEventIdRequest*	*GetTransactionResponse*	Lookup a transaction tree by the ID of an event that appears within it. Returns `NOT_FOUND` if no such transaction exists. For looking up a transaction instead of a transaction tree, please see GetFlatTransactionByEventId
GetTransactionById	*GetTransactionByIdRequest*	*GetTransactionResponse*	Lookup a transaction tree by its ID. Returns `NOT_FOUND` if no such transaction exists. For looking up a transaction instead of a transaction tree, please see GetFlatTransactionById
GetFlatTransactionByEventId	*GetTransactionByEventIdRequest*	*GetFlatTransactionResponse*	Lookup a transaction by the ID of an event that appears within it. Returns `NOT_FOUND` if no such transaction exists.
GetFlatTransactionById	*GetTransactionByIdRequest*	*GetFlatTransactionResponse*	Lookup a transaction by its ID. Returns `NOT_FOUND` if no such transaction exists.
GetLedgerEnd	*GetLedgerEndRequest*	*GetLedgerEndResponse*	Get the current ledger end. Subscriptions started with the returned offset will serve transactions created after this RPC was called.

3.5.1.20 com/digitalasset/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

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

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

3.5.1.21 grpc/health/v1/health_service.proto

HealthCheckRequest

Field	Type	Label	Description
service	*string*		

HealthCheckResponse

Field	Type	Label	Description
status	*HealthCheckResponse.ServingStatus*		

HealthCheckResponse.ServingStatus

Name	Number	Description
UNKNOWN	0	
SERVING	1	
NOT_SERVING	2	

Health

Method name	Request type	Response type	Description
Check	*HealthCheckRequest*	*HealthCheckResponse*	

3.5.1.22 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

3.5.2 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*.

3.5.2.1 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.

3.5.2.2 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: List 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"
    }
  }
}
```

3.5.2.3 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 19: 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 20: 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
          }
        }
      }
    }
  }
}
```

3.5.2.4 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
        }
      }
    }
  }
}
```

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.

3.5.3 Getting started

You can either get the protobufs from Bintray here, or from the `daml` repository here.

3.5.4 Protobuf reference documentation

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

3.5.5 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.

3.5.5.1 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.

3.5.6 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*.

3.5.7 Error handling

Tor 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 with updated Ledger Effective Time (LET) and Maximum Record Time (MRT) values.

INVALID_ARGUMENT The submission failed because of a client error. The platform will definitely reject resubmissions of the same command even with updated LET and MRT values.

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.

3.6 Creating your own bindings

This page gets you started with creating custom bindings for the Digital Asset distributed ledger.

3.6.1 Introduction

Digital Asset currently provides bindings for the following programming languages:

> *Java*
> *Scala*
> *JavaScript (Node.js)*

You can create bindings for any programming language supported by gRPC.

What do we mean by bindings? 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: *The Ledger API using 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)

3.6.2 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.

3.6.2.1 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,
```

(continues on next page)

(continued from previous page)

```
      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())
```

3.6.2.2 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,
```

(continues on next page)

(continued from previous page)

```
      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)
```

3.6.3 Summary

When creating custom bindings for the Digital Asset distributed ledger, 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 *Support* page for how to get in touch with us.

3.6.4 Links

A Scala example that demonstrates how to manually construct ledger commands: https://github.com/digital-asset/daml/tree/master/language-support/scala/examples/iou-no-codegen
A Scala codegen example: https://github.com/digital-asset/daml/tree/master/language-support/scala/examples/quickstart-scala
gRPC documentation: https://grpc.io/docs/
Digital Asset Ledger API gRPC protobuf definitions: https://github.com/digital-asset/daml/tree/master/ledger-api/grpc-definitions

3.7 Application architecture guide

This document is a guide to building applications that interact with a DA ledger deployment (the 'ledger'). It:

describes the characteristics of the ledger API, how this affects the way an application is built (the 'application architecture'), and why it is important to understand this when building applications
describes the resources in the SDK to help with this task
gives some guidelines to help you build correct, performant, and maintainable applications using all of the supported languages

3.7.1 Categories of application

Applications that interact with the ledger normally fall into four categories:

Table 1: Categories of application

Category	Receives transactions?	Sends commands?	Example
Source	No	Yes	An injector that reads new contracts from a file and injects them into the system.
Sink	Yes	No	A reader that pipes data from the ledger into an SQL database.
Automation	Yes	Yes, responding to transactions	Automatic trade registration.
Interactive	Yes (and displays to user)	Yes, based on user input	DA's *Navigator*, which lets you see and interact with the ledger

Additionally, applications can be written in two different styles:

Event-driven - applications base their actions on individual ledger events only.
State-driven - applications base their actions on some model of all contracts active on the ledger.

3.7.1.1 Event-driven applications

Event-driven applications react to events on the the ledger and generate commands and other outputs on a per-event basis. They do not require access to ledger state beyond the event they are reacting to.

Examples are sink applications that read the ledger and dump events to an external store (e.g. an external (reporting) database)

3.7.1.2 State-driven applications

State-driven applications build up a real-time view of the ledger state by reading events and recording contract create and archive events. They then generate commands based on a given state, not just single events.

Examples of these are automation and interactive applications that let a user or code react to complex state on the ledger e.g the DA Navigator tool.

3.7.1.3 Which approach to take

For all except the simplest applications, we generally recommend the state-driven approach. State-driven applications are easier to reason about when determining correctness, so this makes design and implementation easier.

In practice, most applications are actually a mixture of the two styles, with one predominating. It is easier to add some event handling to a state-driven application, so it is better to start with that style.

3.7.2 Structuring an application

Although applications that communicate with the ledger have many purposes, they generally have some common features, usually related to their style: event-driven or state-driven. This section de-

scribes these commonalities, and the major functions of each of these styles.

In particular, all applications need to handle the asynchronous nature of the ledger API. The most important consequence of this is that applications must be multi-threaded. This is because of the asynchronous, separate streams of commands, transaction and completion events.

Although you can choose to do this in several ways, from bare threads (such as a Java Thread) through thread libraries, generally the most effective way of handling this is by adopting a reactive architecture, often using a library such as RxJava.

All the language bindings support this reactive pattern as a fundamental requirement.

3.7.2.1 Structuring event-driven applications

Event-driven applications read a stream of transaction events from the ledger, and convert them to some other representation. This may be a record on a database, some update of a UI, or a differently formatted message that is sent to an upstream process. It may also be a command that transforms the ledger.

The critical thing here is that each event is processed in isolation - the application does not need to keep any application-related state between each event. It is this that differentiates it from a state-driven application.

To do this, the application should:

1. Create a connection to the Transaction Service, and instantiate a stream handler to handle the new event stream. By default, this will read events from the beginning of the ledger. This is usually not what is wanted, as it may replay already processed transactions. In this case, the application can request the stream from the current ledger end. This will, however, cause any events between the last read point and the current ledger end to be missed. If the application must start reading from the point it last stopped, it must record that point and explicitly restart the event stream from there.
2. Optionally, create a connection to the Command Submission Service to send any required commands back to the ledger.
3. Act on the content of events (type, content) to perform any action required by the application e.g writing a database record or generating and submitting a command.

3.7.2.2 Structuring state-driven applications

State-driven applications read a stream of events from the ledger, examine them and build up an application-specific view of the ledger state based on the events type and content. This involves storing some representation of existing contracts on a Create event, and removing them on an Archive event. To be able to remove contract reference, they must be indexed by *contractId*.

This is the most basic kind of update, but other types are also possible. For example, counting the number of a certain type of contract, and establishing relationships between contracts based on business-level keys.

The core of the application is then to write an algorithm that examines the overall state, and generates a set of commands to transform the ledger, based on that state.

If the result of this algorithm depends purely on the current ledger state (and not, for instance, on the event history), you should consider this as a pure function between ledger state and command set, and structure the design of an application accordingly. This is highlighted in the *language bindings*.

To do this, the application should:

1. Obtain the initial state of the ledger by using the Active Contracts service, processing each event received to create an initial application state.
2. Create a connection to the Transaction Service to receive new events from that initial state, and instantiate a stream handler to process them.
3. Create a connection to the Command Submission Service to send commands.
4. Create a connection to the Command Completion Service, and set up a stream handler to handle completions.
5. Read the event stream and process each event to update it's view of the ledger state.
   To make accessing and examining this state easier, this often involves turning the generic description of create contracts into instances of structures (such as class instances that are more appropriate for the language being used. This also allows the application to ignore contract data it does not need.
6. Examine the state at regular intervals (often after receiving and processing each transaction event) and send commands back to the ledger on significant changes.
7. Maintain a record of **pending contracts**: contracts that will be archived by these commands, but whose completion has not been received.
   Because of the asynchronous nature of the API, these contracts will not exist on the ledger at some point after the command has been submitted, but will exist in the application state until the corresponding archive event has been received. Until that happens, the application must ensure that these **pending contracts** are not considered part of the application state, even though their archives have not yet been received. Processing and maintaining this pending set is a crucial part of a state-driven application.
8. Examine command completions, and handle any command errors. As well as application defined needs (such as command re-submission and de-duplications), this must also include handling command errors as described *Common tasks*, and also consider the pending set. Exercise commands that fail mean that contracts that are marked as pending will now not be archived (the application will not receive any archive events for them) and must be returned to the application state.

3.7.2.3 Common tasks

Both styles of applications will take the following steps:

Define an **applicationId** - this identifies the application to the ledger server.
Connect to the ledger (including handling authentication). This creates a client interface object that allows creation of the stream connection described in *Structuring an application*.
Handle execution errors. Because these are received asynchronously, the application will need to keep a record of commands in flight - those send but not yet indicated complete (via an event). Correlate commands and completions via an application-defined *commandId*. Categorize different sets of commands with a *workflowId*.
Handle lost commands. The ledger server does not guarantee that all commands submitted to it will be executed. This means that a command submission will not result in a corresponding completion, and some other mechanism must be employed to detect this. This is done using the values of Ledger Effective Time (LET) and Maximum Record Time (MRT). The server does guarantee that if a command is executed, it will be executed within a time window between the LET and MRT specified in the command submission. Since the value of the ledger time at which a command is executed is returned with every completion, reception of a completion with an record time that is greater than the MRT of any pending command guarantees that the pending command will not be executed, and can be considered lost.
Have a policy regarding command resubmission. In what situations should failing commands be re-submitted? Duplicate commands must be avoided in some situations - what state must be kept to implement this?

Access auxiliary services such as the time service and package service. The *time service* will be used to determine Ledger Effective Time value for command submission, and the package service will be used to determine packageId, used in creating a connection, as well as metadata that allows creation events to be turned in to application domain objects.

3.7.3 Application libraries

We provide several libraries and tools that support the task of building applications. Some of this is provided by the API (e.g. the Active Contracts Service), but mostly is provided by several language binding libraries.

3.7.3.1 Java

The Java API bindings have three levels:

A low-level Data Layer, including Java classes generated from the gRPC protocol definition files and thin layer of support classes. These provide a builder pattern for constructing protocol items, and blocking and non-blocking interfaces for sending and receiving requests and responses.
A Reactive Streams interface, exposing all API endpoints as RxJava Flowables.
A Reactive Components API that uses the above to provide high-level facilities for building state-driven applications.

For more information on these, see the documentation: a *tutorial/description* and the JavaDoc reference.

This API allows a Java application to accomplish all the steps detailed in *Application Structure*. In particular, the Bot abstraction fully supports building of state-driven applications. This is described further in *Architectural Guidance*, below.

3.7.3.2 Scala

The Java libraries above are compatible with Scala and can be used directly.

3.7.3.3 gRPC

We provides the full details of the gRPC service and protocol definitions. These can be compiled to a variety of target languages using the open-source protobuf and gRPC tools. This allows an application to attach to an interface at the same level as the provided Data Layer Java bindings.

3.7.4 Architecture guidance

This section presents some suggestions and guidance for building successful applications.

3.7.4.1 Use a reactive architecture and libraries

In general, you should consider using a reactive architecture for your application. This has a number of advantages:

It matches well to the streaming nature of the ledger API.
It will handle all the multi-threading issues, providing you with sequential model to implement your application code.
It allows for several implementation strategies that are inherently scalable e.g RxJava, Akka Streams/Actors, RxJS, RxPy etc.

3.7.4.2 Prefer a state-driven approach

For all but the simplest applications, the state-driven approach has several advantages:

> It's easier to add direct event handling to state-driven applications than the reverse.
> Most applications have to keep some state.
> DigitalAsset language bindings directly support the pattern, and provide libraries that handle many of the required tasks.

3.7.4.3 Consider a state-driven application as a function of state to commands

As far as possible, aim to encode the core application as a function between application state and generated commands. This helps because:

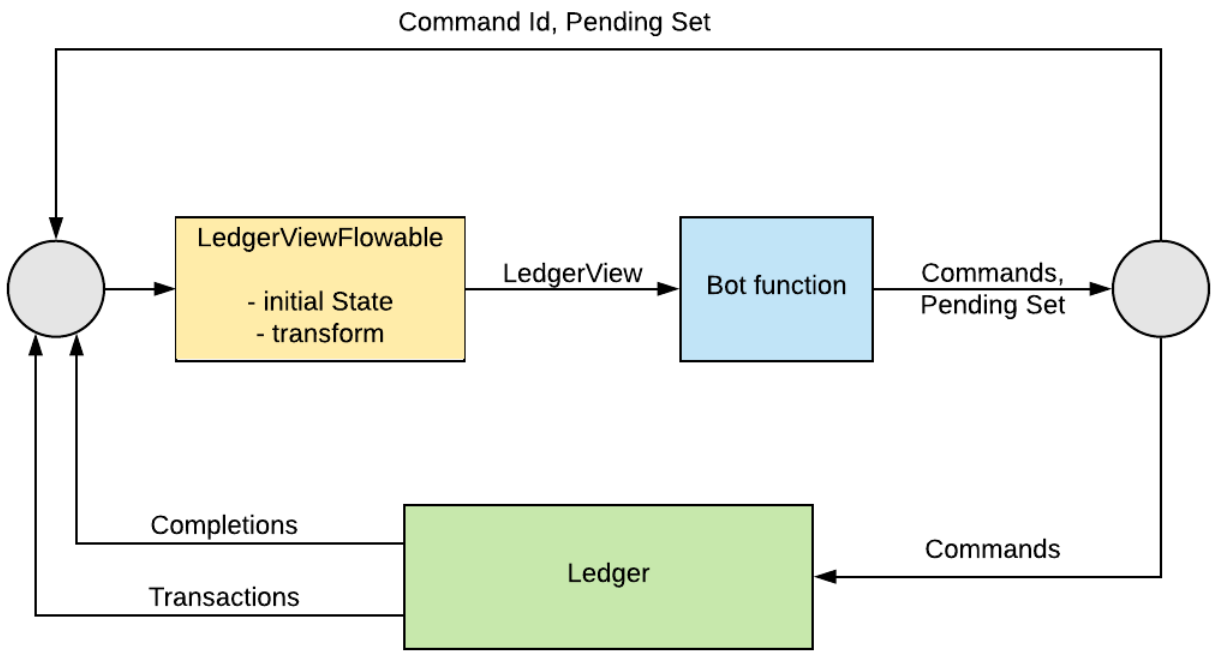
> It separates the application into separate stages of event transformation, state update and command generation.
> The command generation is the core of the application - implementing as a pure function makes it easy to reason about, and thus reduces bugs and fosters correctness.
> Doing this will also require that the application is structured so that the state examined by that function is stable - that is, not subject to an update while the function is running. This is one of the things that makes the function, and hence the application, easier to reason about.

The Java Reactive Components library provides an abstraction and framework that directly supports this. It provides a Bot abstraction that handles much of work of doing this, and allows the command generation function to be represented as an actual Java function, and wired into the framework, along with a transform function that allows the state objects to be Java classes that better represent the underlying contracts.

This allows you to reduce the work of building and application to the tasks of:

> defining the Bot function.
> defining the event transformation.
> defining setup tasks such as disposing of command failure, connecting to the ledger and obtaining ledger- and package- IDs.

The framework handles much of the work of building a state-driven application. It handles the streams of events and completions, transforming events into domain objects (via the provided event transform function) and storing them in a LedgerView object. This is then passed to the Bot function (provided by the application), which generates a set of commands and a pending set. The commands are sent back to the ledger, and the pending set, along with the commandId that identifies it, is held by the framework (LedgerViewFlowable). This allows it to handle all command completion events.



Full details of the framework are available in the links described in the *Java library* above.

3.7.5 Commonly used types

Primitive and structured types (records, variants and lists) appearing in the contract constructors and choice arguments are compatible with the types defined in the current version of DAML-LF (v1). They appear in the submitted commands and in the event streams.

There are some identifier fields that are represented as strings in the protobuf messages. They are opaque: you shouldn't interpret them in client code, except by comparing them for equality. They include:

Transaction IDs
Event IDs
Contract IDs
Package IDs (part of template identifiers)

There are some other identifiers that are determined by your client code. These aren't interpreted by the server, and are transparently passed to the responses. They include:

Command IDs: used to uniquely identify a command and to match it against its response.
Application ID: used to uniquely identify client process talking to the server. You could use a combination of submitting party, command ID, and application ID for deduplication of commands.
Workflow IDs: identify chains of transactions. You can use these to correlate transactions sent across time spans and by different parties.

3.7.5.1 Testing

Testing is fundamental to ensure correctness and improve maintainability.

Testing is usually divided into different categories according to its scope and aim:

unit testing verifies single properties of individual components
integration testing verifies that an aggregation of components behaves as expected

acceptance testing checks that the overall behavior of a whole system satisfies certain criteria

Both tests in the small scale (unit testing) and large (acceptance testing) tend to be specific to the given component or system under test.

This chapter focuses on providing portable approaches and techniques to perform integration testing between your components and an actual running ledger.

3.7.6 Test the business logic with a ledger

In production, your application is going to interact with a DAML model deployed on an actual ledger. Each model is usually specific to a business need and describes specific workflows.

Mocking a ledger response is usually not desirable to test the business logic, because so much of it is encapsulated in the DAML model. This makes integration testing with an actual running ledger fundamental to evaluating the correctness of an application.

This is usually achieved by running a ledger as part of the test process and run several tests against it, possibly coordinated by a test framework. Since the in-memory sandbox shipped as part of the SDK is a full-fledged implementation of a DAML ledger, it's usually the tool of choice for these tests. Please note that this does not replace acceptance tests with the actual ledger implementation that your application aims to use in production. Whatever your choice is, sharing a single ledger to run several tests is a suggested best practice.

3.7.7 Share the ledger

Sharing a ledger is useful because booting a ledger and loading DAML code into it takes time. Since to properly test your application you need an actual ledger you're likely to have a lot of very short tests whose total running time would suffer from the overhead of running a new ledger for every test.

Tests must thus be designed to not interfere with each other. Both the transaction and the active contract service offer the possibility of filtering by party. Parties can thus be used as a way to isolate tests.

You can use the party management service to allocate new parties and use them to test your application. You can also limit the number of transactions read from the ledger by reading the current offset of the ledger end before the test starts, since no transactions can possibly appear for the newly allocated parties before this time.

In summary:

1. retrieve the current offset of the ledger end before the test starts 1. use the party management service to allocate the parties needed by the test 1. whenever you issue a command, issue it as one of the parties allocated for this test 1. whenever you need to get the set of active contracts or a stream of transactions, always filter by one or more of the parties allocated for this test

This isolation between instances of tests also means that different tests can be run completely in parallel with respect to each other, possibly improving on the overall running time of your test suite.

3.7.8 Reset if you need to

It may be the case that you are running a very high number of tests, verifying the ins and outs of a very complex application interacting with an equally complex DAML model.

If that's the case, the leak of resources caused by the approach to test isolation mentioned above can become counterproductive, causing slow-downs or even crashes as the ledger backing your test

suite has to keep track of more parties and more transactions that are actually no longer relevant after the test itself finishes.

As a last resort for these cases, your tests can use the reset service, which ledger implementations can optionally expose for testing.

The reset service has a single `reset` method that will cause all the accumulated state to be dropped, including all active contracts, the entire history of transactions and all allocated users. Only the DAML packges loaded in the ledger is preserved, possibly saving on the time needed to be loaded as opposed to simply spinning up a new ledger.

The reset service momentarily shuts down the gRPC channel it communicates over, so your testing infrastructure must take this into account and, when the `reset` is invoked, must ensure that tests are temporarily suspended as attempts to reconnect with the rebooted ledger are performed. There is no guarantee as to how long the reset will take, so this should also be taken into account when attempting to reconnect.

Chapter 4

SDK tools

4.1 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>`
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 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.

4.1.1 Moving to the `daml` assistant

To move your projects to use `daml`, and see the difference between `da` commands and `daml` commands, read the *Moving to the new DAML assistant*.

4.1.2 Full help for commands

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

4.1.3 Configuration files

The DAML assistant and the DAML 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 DAML SDK builds and interacts with the project

4.1.3.1 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.

Here is an example `daml-config.yaml`:

```
auto-install: true
update-check: 86400
```

4.1.3.2 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. It incorporates info from `da.yaml` in the generated `daml.yaml`, if `da.yaml` is available (see *Moving to the new DAML assistant*).

`daml new` creates a skeleton application in a new project folder, which includes a config file. For example, `daml new my_project` creates a new folder `my_project` with a project config file `daml.yaml` like this:

```
sdk-version: __VERSION__
name: __PROJECT_NAME__
source: daml
scenario: Main:setup
parties:
  - Alice
  - Bob
version: 1.0.0
exposed-modules:
```

(continues on next page)

(continued from previous page)

```
  - Main
dependencies:
  - daml-prim
  - daml-stdlib
scenario-service:
  grpc-max-message-size: 134217728
  grpc-timeout: 60
build-options: ["--ghc-option", "-Werror",
                "--ghc-option", "-v"]
```

Here is what each field means:

`sdk-version`: the SDK version that this project uses.

The assistant automatically downloads and installs this version if needed (see the `auto-install` setting in the global config). We recommend keeping this up to date with the latest stable release of the SDK. It is possible to override the version without modifying the `daml.yaml` file by setting the `DAML_SDK_VERSION` environment variable. This is mainly useful when you are working with an external project that you want to build with a specific version.

The assistant will warn you when it is time to update this setting (see the `update-check` setting in the global config to control how often it checks, or to disable this check entirely).

`name`: the name of the project. This determines the filename of the `.dar` file compiled by `daml build`.

`source`: the root folder of your DAML source code files relative to the project root.

`scenario`: the name of the scenario 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`: the dependencies of this project.

`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.

`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.

4.1.4 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
```

4.1.5 Managing SDK releases

In general the `daml` assistant will install versions and guide you when you need to update SDK versions or project settings. If you disable `auto-install` and `update-check` in the global config file, you will have to manage SDK releases manually.

To download and install the latest stable SDK release and update the assistant, run:

```
daml install latest --activate
```

Remove the `--activate` flag if you only want to install the latest release without updating the `daml` assistant in the process. If it is already installed, you can force reinstallation by passing the `--force` flag. See `daml install --help` for a full list of options.

To install the SDK release specified in the project config, run:

```
daml install project
```

To install a specific SDK version, for example version `0.12.17`, run:

```
daml install 0.12.17
```

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
```

4.2 DAML Sandbox

The DAML Sandbox, or Sandbox for short, is a simple ledger implementation that enables rapid application prototyping by simulating a Digital Asset Distributed Ledger.

You can start Sandbox together with *Navigator* using the `daml start` command in a DAML SDK 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. Sandbox will also be given the name of the startup scenario specified in the project's `daml.yaml`. Finally, it launches the navigator connecting it to the running Sandbox.

It is possible to execute the Sandbox launching step in isolation by typing `daml sandbox`.

Sandbox can also be run manually as in this example:

```
$ daml sandbox Main.dar --scenario Main:example


  ____                ____
 / __/__ ____   ___/ / /   ___ __ __
```

(continues on next page)

(continued from previous page)

```
 _\ \/ _ `/ _ \/ _   / _ \/ _ \\ \ /
/___/\_,_/_//_/\_,_/_.__/\___/_\_\
initialized sandbox with ledger-id = sandbox-16ae201c-b2fd-45e0-af04-
↪c61abe13fed7, port = 6865,
dar file = DAR files at List(/Users/donkeykong/temp/da-sdk/test/Main.dar),
↪time mode = Static, daml-engine = {}
Initialized Static time provider, starting from 1970-01-01T00:00:00Z
listening on localhost:6865
```

Here, `daml sandbox` tells the SDK Assistant to run `sandbox` from the active SDK release and pass it any arguments that follow. The example passes the DAR file to load (`Main.dar`) and the optional `--scenario` flag tells Sandbox to run the `Main:example` scenario on startup. The scenario must be fully qualified; here `Main` is the module and `example` is the name of the scenario, separated by a `:`.

Note: The scenario is used for testing and development only, and is not supported by production DAML Ledgers. It is therefore unadvisable to rely on scenarios for ledger initialization.

`submitMustFail` is only supported by the test-ledger used by `daml test` and the IDE, not by the Sandbox.

4.2.1 Running with persistence

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

4.2.2 Command-line reference

To start Sandbox, run: `sandbox [options] <archive>...`

To see all the available options, run `daml sandbox --help`

4.3 Navigator

The Navigator is a front-end that you can use to connect to any Digital Asset 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 DAML SDK. Refer to *Advanced usage* for information on using Navigator outside the context of the SDK.

4.3.1 Navigator functionality

Connect Navigator to any Digital Asset 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.)

4.3.2 Installing and starting Navigator

Navigator ships with the DAML SDK. To launch it:

1. Start Navigator via a terminal window running *SDK 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

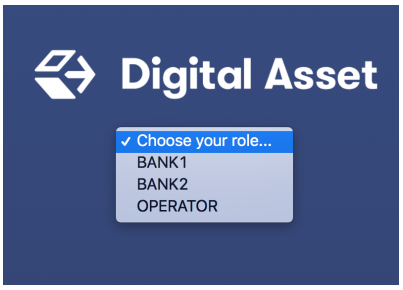
   When running `daml start` you will see the Navigator URL. By default it will be http://localhost:7500/.

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.

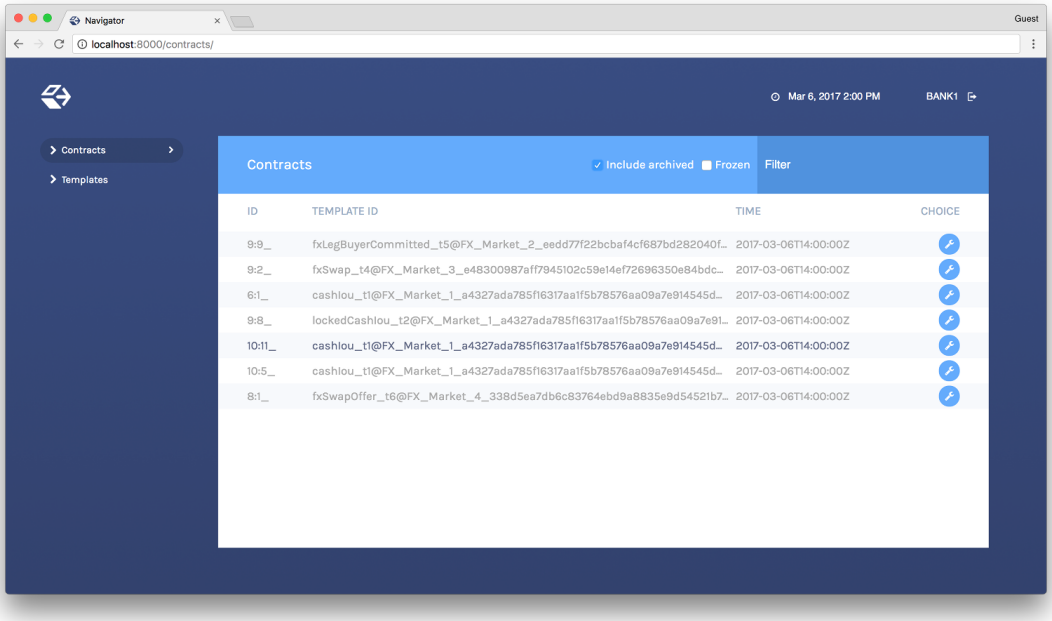
4.3.3 Choosing a party / changing the party

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.



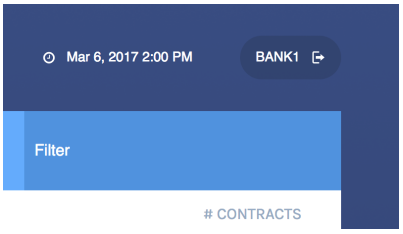
Note: The party choices are configured on startup. (Refer to *DAML Assistant (daml)* or *Advanced usage* for more instructions.)

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.

4.3.4 Logging out

To log out, click the name of the current party in the top-right corner of the screen.

4.3.5 Viewing templates or contracts

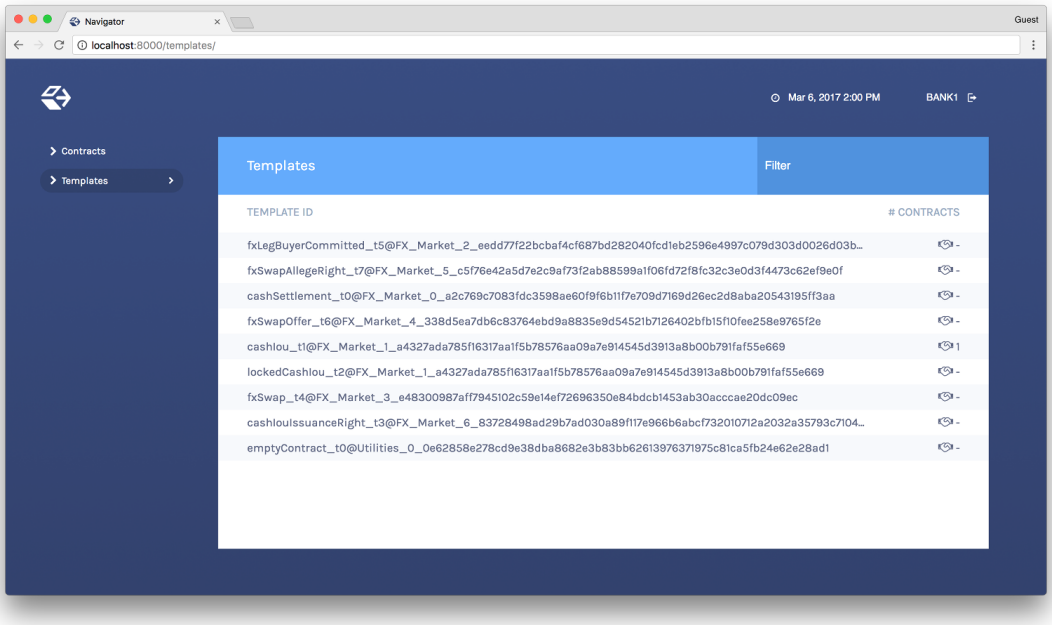
DAML *contract templates* are models that contain the agreement statement, all the applicable parameters, and the choices that can be made in acting on that data. They specify acceptable input and

the resulting output. A contract template contains placeholders rather than actual names, amounts, dates, and so on. In a *contract instance*, 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.

4.3.5.1 Listing templates

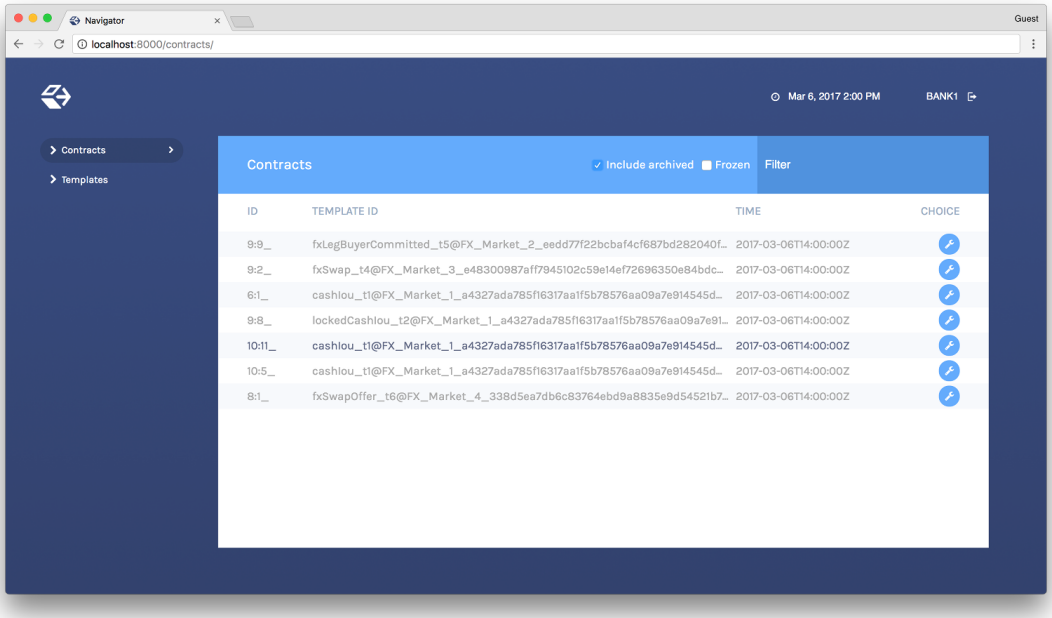
To see what contract templates are available on the ledger you are connected to, choose **Templates** in the left pane of the main Navigator screen.



Use the **Filter** field at the top right to select template IDs that include the text you enter.

4.3.5.2 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.

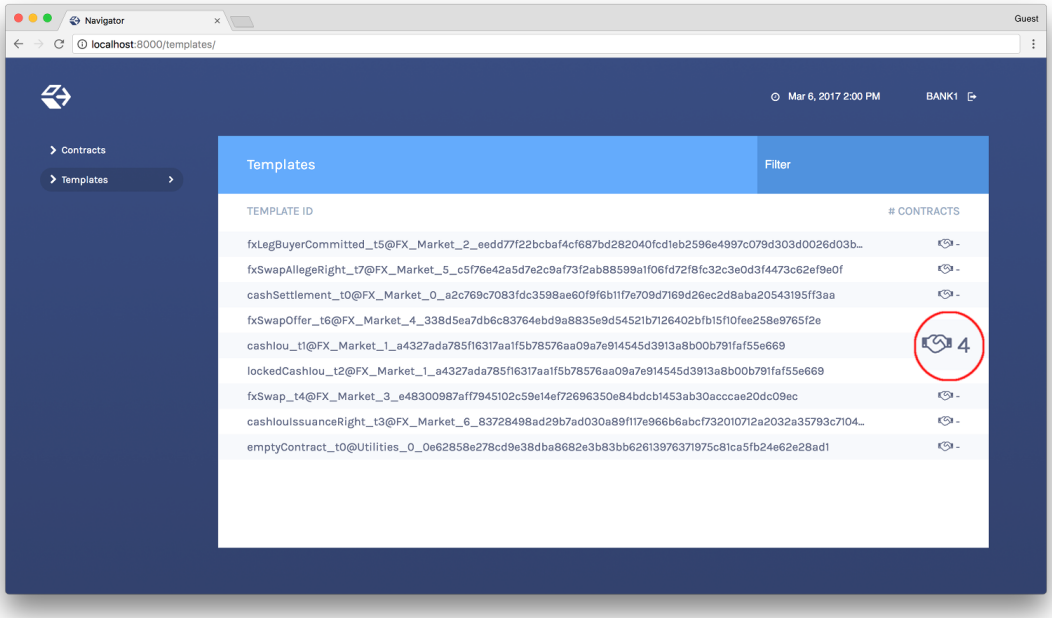
Filter the displayed contracts by entering text in the **Filter** field at the top right.
Use the **Include Archived** checkbox at the top to include or exclude archived contracts.

4.3.5.3 Viewing contracts based on a template

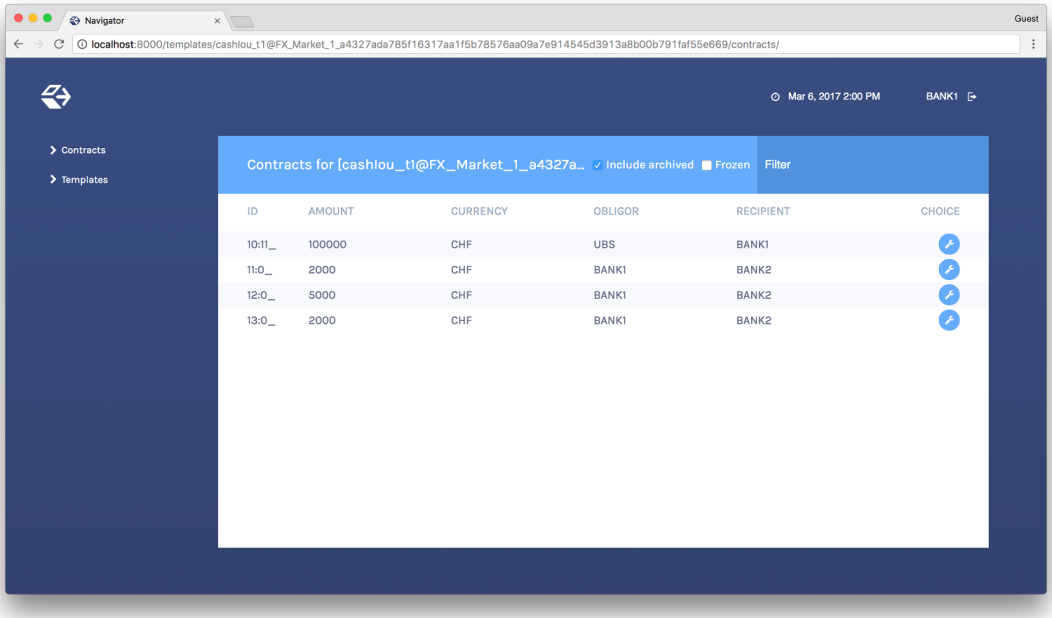
You can also view the list of contracts that are based on a particular template.

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



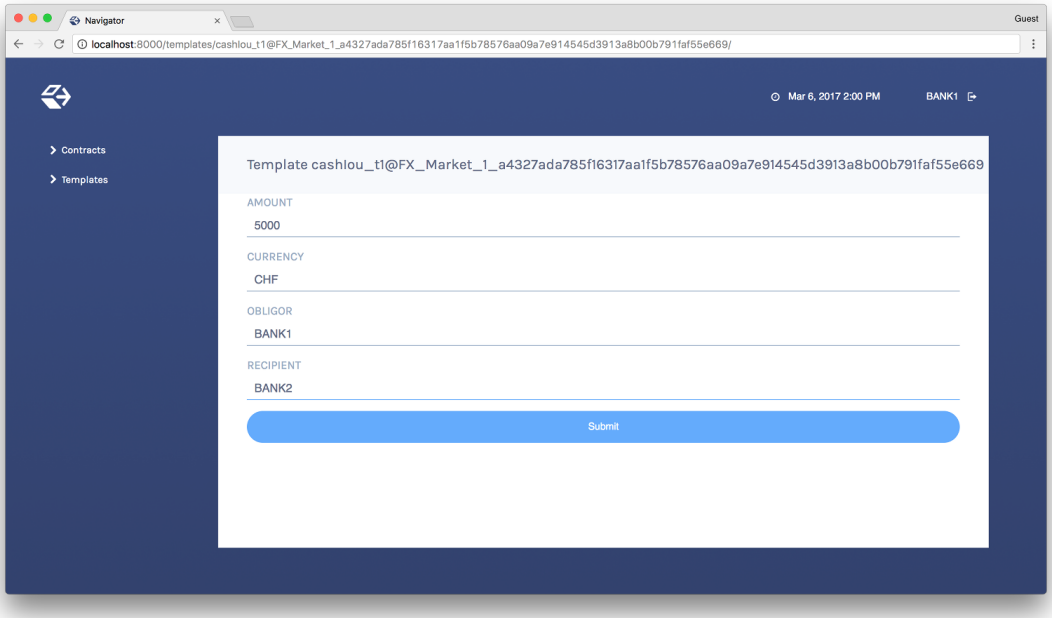
List of Contracts



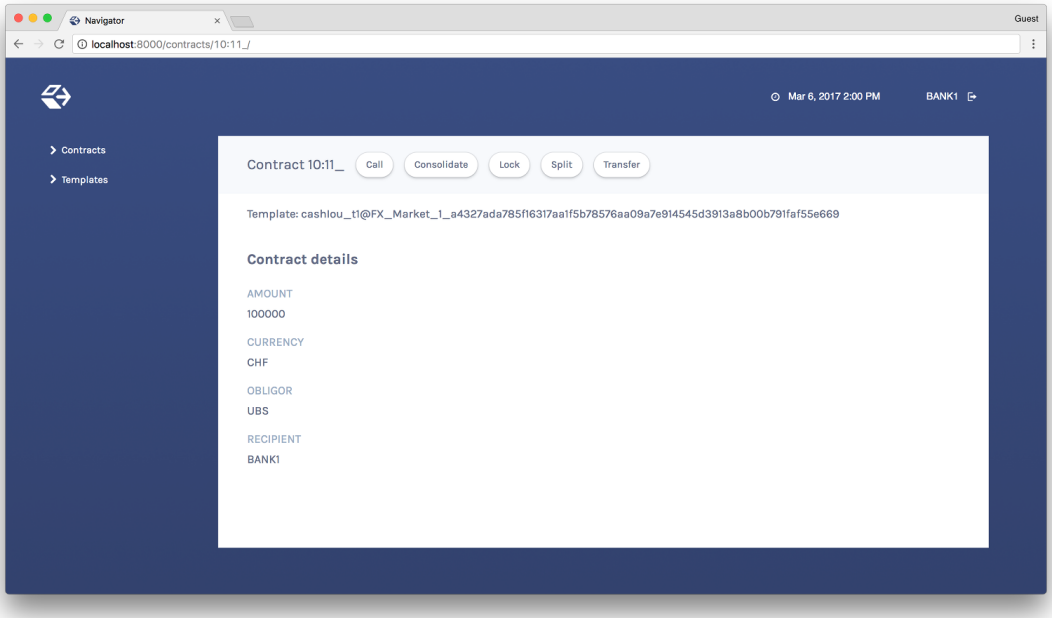
4.3.5.4 Viewing template and contract details

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



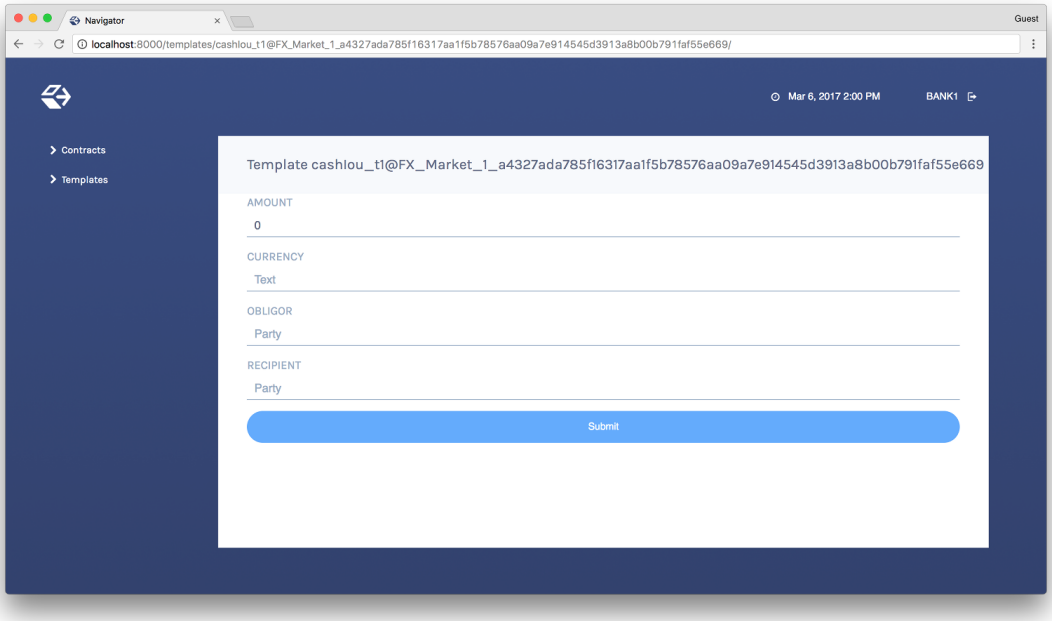
4.3.6 Using Navigator

4.3.6.1 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 experimenting during development.

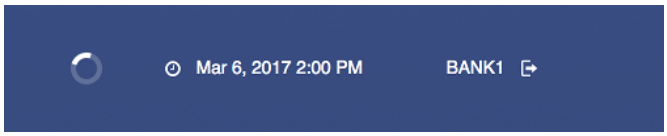
To create a contract based on a template:

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

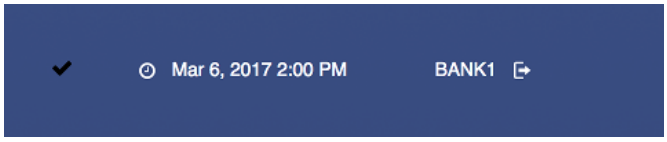


When the command has been committed to the ledger, the loading indicator in the navbar at the top will display a tick mark.

While loading



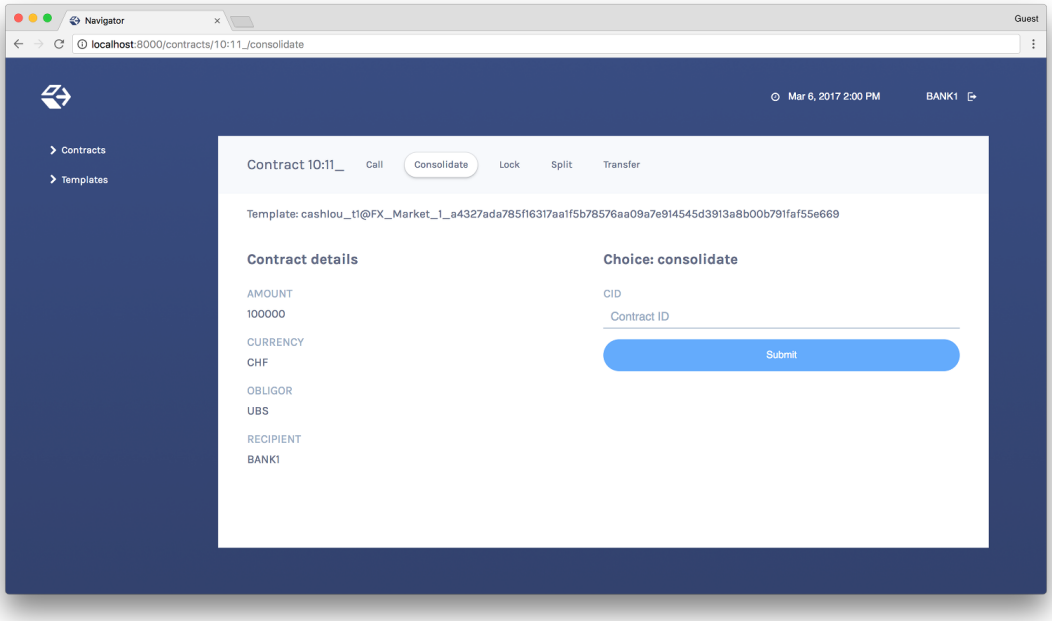
When committed to the ledger



4.3.6.2 Exercising choices

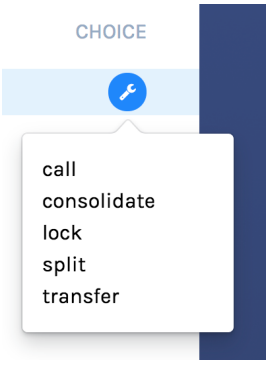
To exercise a choice:

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.



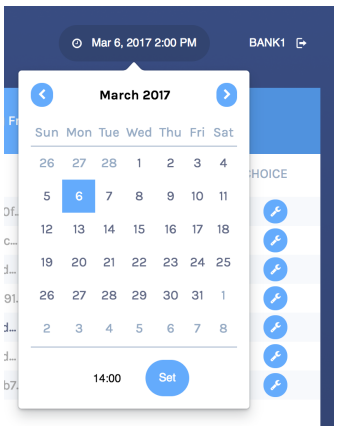
You will see the loading and confirmation indicators, as pictured above in Creating Contracts.

4.3.6.3 Advancing time

It is possible to advance time against the DAML Sandbox. (This is not true of the Digital Asset ledger.) 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.



4.3.7 Authenticating Navigator

If you are running Navigator against a Ledger API server that requires authentication, 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 in two ways:

store it in the file `$DAML_HOME/secret`, where `$DAML_HOME` is `$HOME/.daml` on Linux and MacOS, and `%APPDATA%/daml` on Windows, or
store it in a different file and provide the path to it using the `-t` or `--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:

4.3.8 Advanced usage

4.3.8.1 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',
```

(continues on next page)

(continued from previous page)

```
  major: 2,
  minor: 0,
};

export const customViews = (userId, party, role) => ({
  customview1: {
    type: "table-view",
    title: "Filtered contracts",
    source: {
      type: "contracts",
      filter: [
        {
          field: "id",
          value: "1",
        }
      ],
      search: "",
      sort: [
        {
          field: "id",
          direction: "ASCENDING"
        }
      ]
    },
    columns: [
      {
        key: "id",
        title: "Contract ID",
        createCell: ({rowData}) => ({
          type: "text",
          value: rowData.id
        }),
        sortable: true,
        width: 80,
        weight: 0,
        alignment: "left"
      },
      {
        key: "template.id",
        title: "Template ID",
        createCell: ({rowData}) => ({
          type: "text",
          value: rowData.template.id
        }),
        sortable: true,
        width: 200,
        weight: 3,
        alignment: "left"
      }
```

(continues on next page)

(continued from previous page)

```
    ]
  }
})
```

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).

4.3.8.2 Using Navigator outside the SDK

This section explains how to work with the Navigator if you have a project created outside of the normal SDK workflow and want to use the Navigator to inspect the ledger and interact with it.

Note: If you are using the Navigator as part of the DAML SDK, you do not need to read this section.

The Navigator is released as a fat Java *.jar* file that bundles all required dependencies. This JAR is part of the SDK release and can be found using the SDK Assistant's `path` command:

```
da path navigator
```

Use the `run` command to launch the Navigator JAR and print usage instructions:

```
da run navigator
```

Arguments may be given at the end of a command, following a double dash. For example:

```
da run navigator -- server \
  --config-file my-config.conf \
  --port 8000 \
  localhost 6865
```

The Navigator requires a configuration file specifying each user and the party they act as. It has a `.conf` ending by convention. The file follows this form:

```
users {
    <USERNAME> {
        party = <PARTYNAME>
    }
    ..
}
```

In many cases, a simple one-to-one correspondence between users and their respective parties is sufficient to configure the Navigator. Example:

```
users {
    BANK1 { party = "BANK1" }
    BANK2 { party = "BANK2" }
    OPERATOR { party = "OPERATOR" }
}
```

4.3.8.3 Using Navigator with the Digital Asset ledger

By default, Navigator is configured to use an unencrypted connection to the ledger. To run Navigator against a secured Digital Asset 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
```

Chapter 5

Background concepts

5.1 Glossary of concepts

5.1.1 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.

5.1.1.1 Contract, contract instance

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*.

They're sometimes referred to as a **contract instance** to make clear that this is an instantiated contract, as opposed to a *template*.

Active contract, archived contract

When a *contract* is created on a *ledger*, it becomes **active**. But that doesn't mean it will stay active forever: it can be **archived**. This can happen:

> if the *signatories* of the contract decide to archive it
> if a *consuming choice* is exercised on the contract

Once the contract is archived, it is no longer valid, and *choices* on the contract can no longer be exercised.

5.1.1.2 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*.

5.1.1.3 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 instance*.

Disjunction choice, flexible controllers

A **disjunction choice** has more than one *controller*.

If a contract uses **flexible controllers**, this means you don't specify the controller of the *choice* at *creation* time of the *contract*, but at *exercise* time.

5.1.1.4 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 instance*. 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 instance*. 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 instance*.

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*.

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 instance*.

Maintainer

The **maintainer** is a *party* that is part of a *contract key*. They must always be a *signatory* on the *contract* that they maintain the key for.

It's not possible for keys to be globally unique, because there is no party that will necessarily know about every contract. However, by including a party as part of the key, this ensures that the maintainer *will* know about all of the contracts, and so can guarantee the uniqueness of the keys that they know about.

For documentation on contract keys, see *Contract keys*.

5.1.1.5 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 instance
`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 instance
`fetchByKey`	`k -> Update (ContractId c, c)`	Same as `fetch`
`lookupByKey`	`k -> Update (Optional (ContractId c))`	All key maintainers

At runtime, the DAML execution engine computes the required authorizing parties from this mapping. It also computes which parties have given authorization to the update in question. A party is giving authorization to an update in one of two ways:

It is the signatory of the contract that contains the update action.
It is element of the controllers executing the choice containing the update action.

Only if all required parties have given their authorization to an update action, the update action is well-authorized and therefore executed. A missing authorization leads to the abortion of the update action and the failure of the containing transaction.

It is noteworthy, that authorizing parties are always determined only from the local context of a choice in question, that is, its controllers and the contract's signatories. Authorization is never inherited from earlier execution contexts.

5.1.1.6 Standard library

The **DAML standard library** is a set of *DAML* functions, classes and more that make developing with DAML easier.

For documentation, see /daml/reference/base.

5.1.1.7 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 evaulated programmatically.

See *Reference: templates*.

5.1.1.8 Create

A **create** is an update that creates a *contract instance* on it the *ledger*.

Contract creation requires *authorization* from all its *signatories*, or the create will fail. For how to get authorization, see the *propose-accept* and *multi-party agreement* patterns.

A *party submits* a create *command*.

See *Reference: updates*.

5.1.1.9 Exercise

An **exercise** is an action that exercises a *choice* on a *contract instance* on the *ledger*. If the choice is *consuming*, the exercise will *archive* the contract instance; if it is *nonconsuming*, the contract instance will stay active.

Exercising a choice requires *authorization* from all of the *controllers* of the choice.

A *party submits* an exercise *command*.

See *Reference: updates*.

5.1.1.10 Scenario

A **scenario** is a way of testing DAML code during development. You can run scenarios 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*
for making sure that parties can create contracts, observe contracts, and exercise choices (and that they CANNOT create contracts, observe contracts, or exercise choices that they should not be able to)
acting as DAML unit tests to confirm that everything keeps working correctly

Scenarios emulate a real 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 scenario fails) what caused it to fail.

See *Testing using scenarios*.

5.1.1.11 Contract key

A **contract key** allows you to uniquely identify a *contract instance* of a particular *template*, similarly to a primary key in a database table.

A contract key requires a *maintainer*: a simple key would be something like a tuple of text and maintainer, like `(accountId, bank)`.

See *Contract keys*.

5.1.1.12 DAR file, DALF file

A `.dar` file is the result of compiling DAML using the *Assistant*. Its underlying format is *DAML-LF*.

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.

5.1.2 SDK tools

5.1.2.1 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 SDK tools, and download new SDK versions.

See *DAML Assistant (daml)*.

5.1.2.2 Studio

DAML Studio is a plugin for Visual Studio Code, and is the IDE for writing DAML code.

See *DAML Studio*.

5.1.2.3 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*.

5.1.2.4 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*.

Navigator Console

This is the version of Navigator that runs on the command-line. It has similar functionality to the GUI.

See *Navigator Console*.

5.1.2.5 Extractor

Extractor is a tool for extracting contract data for a single party into a PostgreSQL database.

See *Extractor*.

5.1.3 Building applications

5.1.3.1 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 *Writing applications using the Ledger API* page.

5.1.3.2 Ledger API

The **Ledger API** is an API that's exposed by any *DAML ledger*. 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*.

5.1.3.3 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*.

Node.js bindings

An idiomatic JavaScript library for writing *ledger applications*. See *Node.js bindings*.

Scala bindings

An idiomatic Scala library for writing *ledger applications*. See *Java bindings*.

gRPC API

The low-level ledger API that all of the other bindings use. Written in gRPC. See *The Ledger API using gRPC*.

5.1.3.4 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.

5.1.3.5 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*.

5.1.3.6 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 inside the DAML SDK, it's used for:

executing DAML code on the Sandbox or on another platform
sending and receiving values via the Ledger API (using a protocol such as gRPC)
generating code in other languages for interacting with DAML models (often called codegen)

5.1.4 General concepts

5.1.4.1 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 *DA Ledger Model* page.

When you're developing, you'll use *Sandbox* as your ledger.

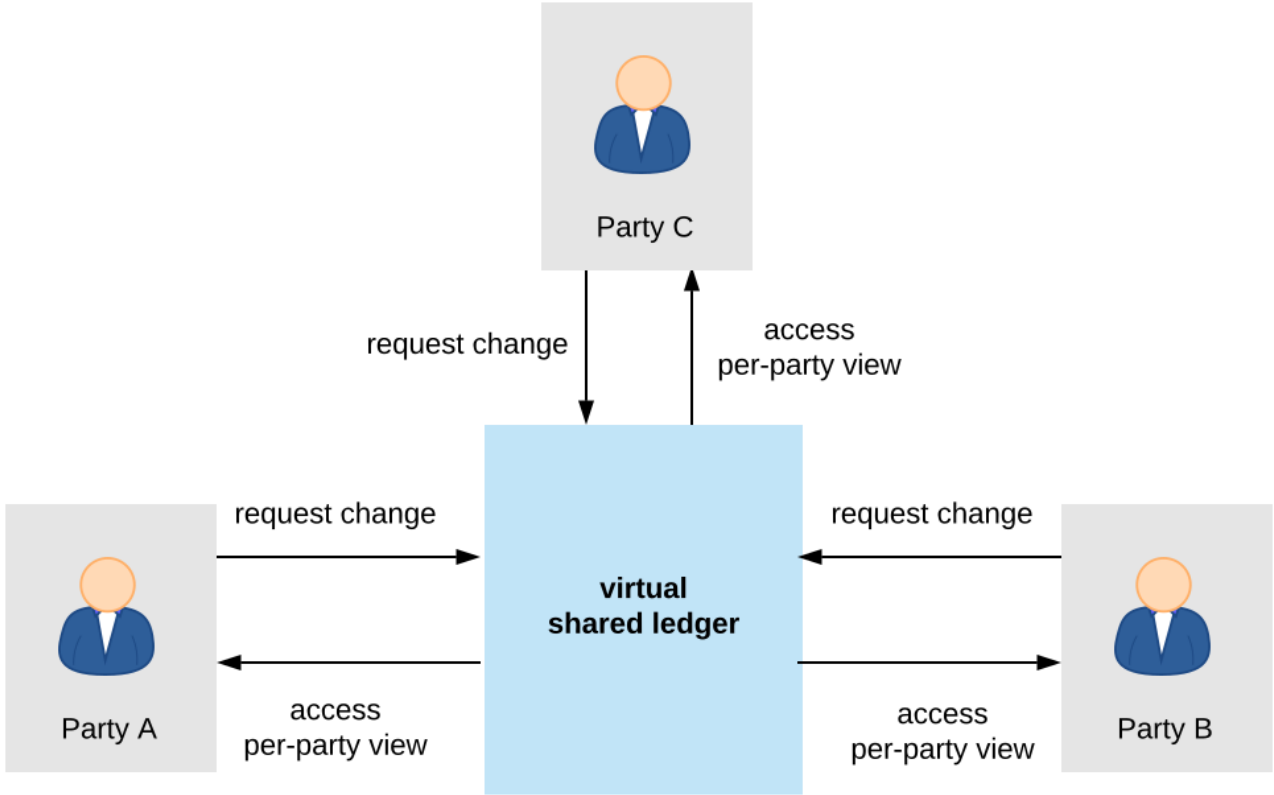
If you would like to integrate DAML with a storage infrastructure not already in development (see daml.com), please get in touch on Slack in the channel `#daml-contributors`.

5.1.4.2 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.

5.2 DA Ledger Model

The Digital Asset Platform enables 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 DA ledger model defines:

1. what the ledger looks like - the structure of DA ledgers
2. who can request which changes - the integrity model for DA ledgers
3. who sees which changes and data - the privacy model for DA ledgers

The below sections review these concepts of the ledger model in turn. They also briefly describe the link between DAML and the model.

5.2.1 Structure

This section looks at the structure of a DA ledger and the associated ledger changes. The basic building blocks of changes are *actions*, which get grouped into *transactions*.

5.2.1.1 Actions and Transactions

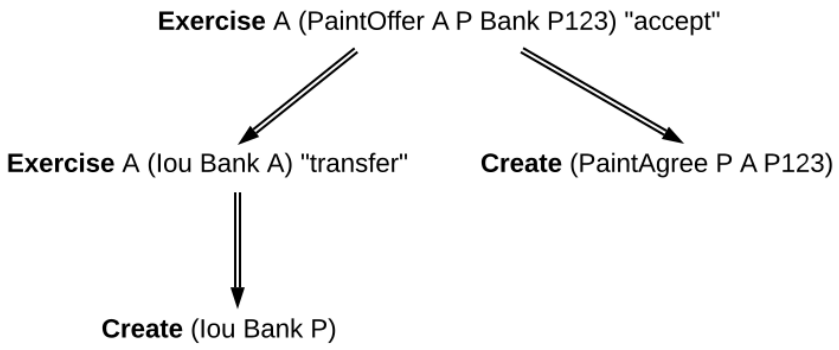
One of the main features of the DA ledger model is a *hierarchical action structure*.

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, transfering 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 DA 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, the painter should not be able to accept the 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 in force 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.

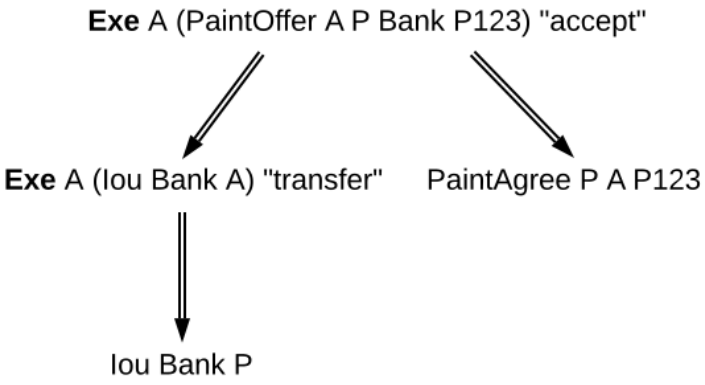
```
Action        ::= 'Create' contract
                | 'Exercise' party* contract Kind Transaction
                | 'Fetch' party* contract
                | 'NoSuchKey' key
Transaction   ::= Action*
Kind          ::= 'Consuming' | 'NonConsuming'
```

The visual notation presented earlier captures actions precisely with conventions that:

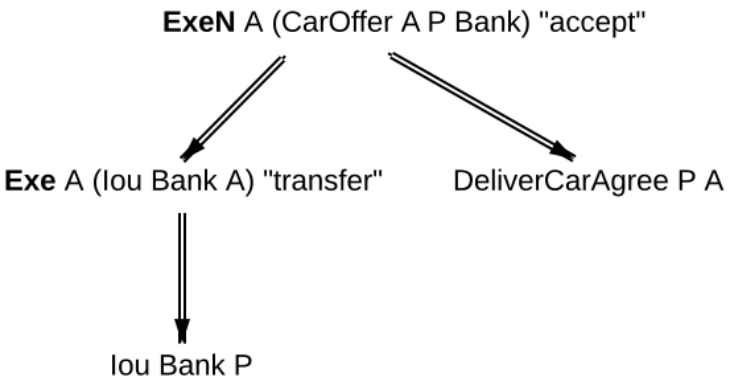
1. **Exercise** denotes consuming, **ExerciseN** non-consuming exercises, and **Fetch** a fetch.

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.

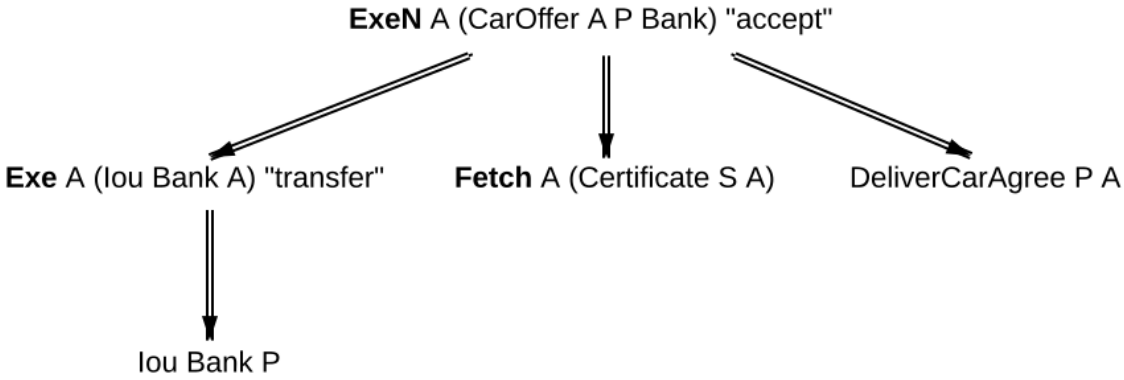
An alternative shorthand notation, shown below uses the abbreviations **Exe** and **ExeN** for exercises, and omits the **Create** labels on create actions.



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:

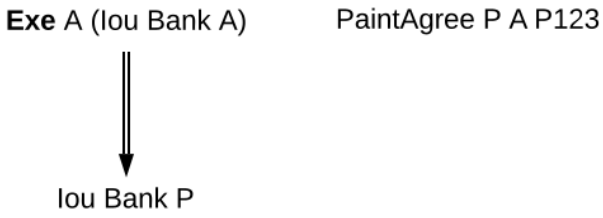
PaintOffer houseOwner painter obligor refNo Intuitively an offer (with a reference number) by which the painter proposes to the house owner to paint her house, in exchange for a single IOU token issued by the specified obligor.

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).

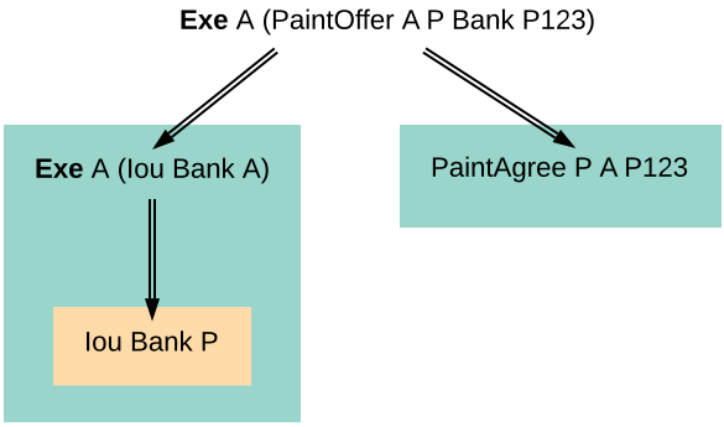
In practice, multiple IOU contracts would exist between the same *obligor* and *owner*, in which case each contract should have a unique identifier. However, in this section, each contract only appears once, allowing us to drop the notion of identifiers for simplicity reasons.

A **transaction** is a list of actions. Thus, the consequences of an exercise form a transaction. In the example, the consequences of the Alice's exercise form the following transaction, where actions are again ordered left-to-right.

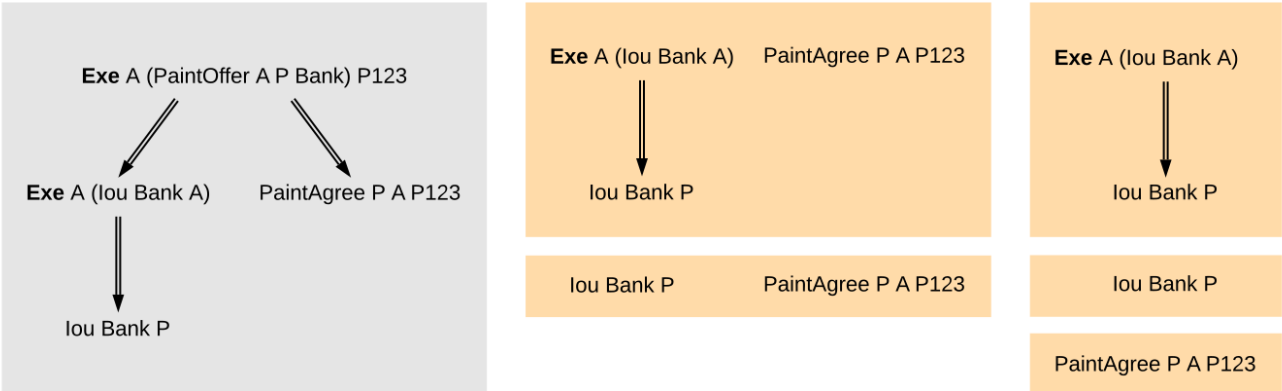


For an action *act*, its **proper subactions** are all actions in the consequences of *act*, together with all of their proper subactions. Additionally, *act* is a (non-proper) **subaction** of itself.

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 the transaction consisting of just one action, the paint offer acceptance, the image below shows all its proper subtransactions on the right (yellow boxes).



To illustrate *contract keys*, suppose that the contract key for a *PaintOffer* consists of the reference number and the painter. So Alice can refer to the *PaintOffer* by its key *(P, P123)*. To make this explicit, we use the notation *PaintOffer @P A &P123* for contracts, where @ and & mark the parts that belong to a key. (The difference between @ and & will be explained in the *integrity section*.) The ledger integrity constraints in the next section ensure that there is always at most one active *PaintOffer* for a given key. So if the painter retracts its *PaintOffer* and later Alice tries to accept it, she can then record the absence with a *NoSuchKey (P, P123)* key assertion.

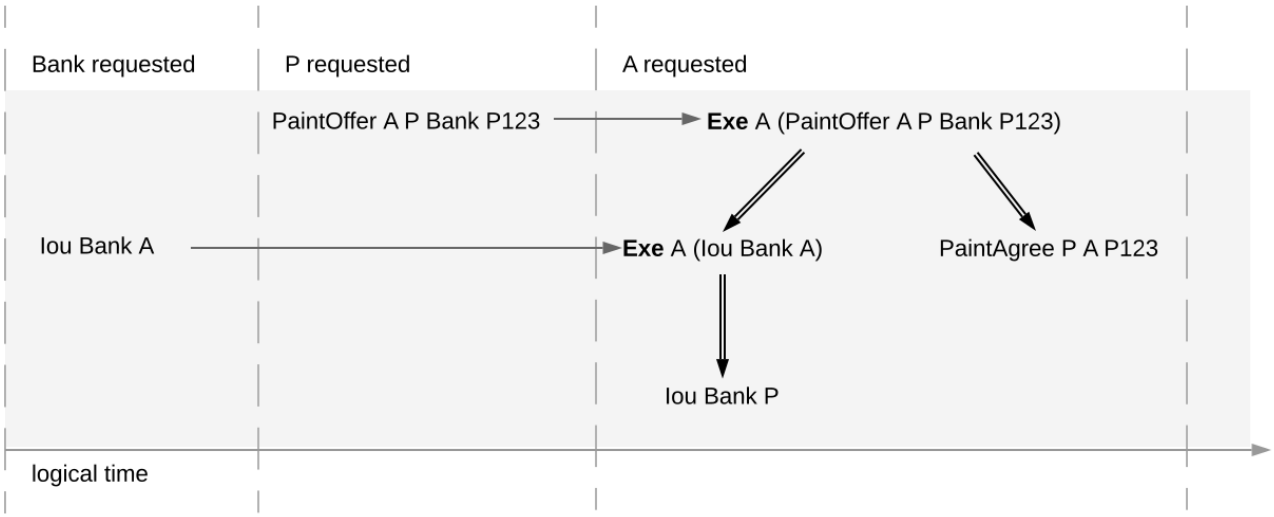
5.2.1.2 Ledgers

The transaction structure records the contents of the changes, but not *who requested them*. This information is added by the notion of a **commit**: a transaction paired with the parties that requested it, called the **requesters** of the commit. In the ledger model, a commit is allowed to have multiple requesters, although the current DA Platform API offers the request functionality only to individual parties. Given a commit *(p, tx)* with transaction $tx = act_1, , act_n$, 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 DA 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 DA 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.

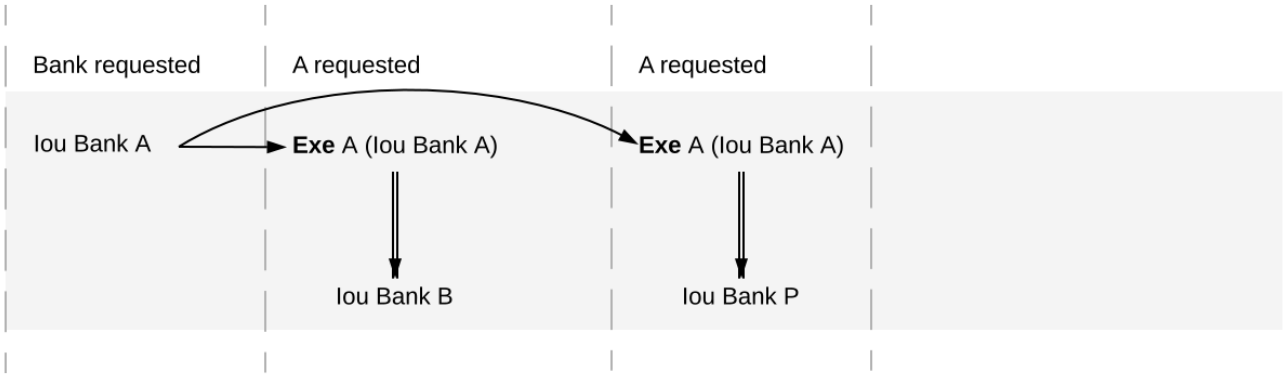


Fig. 1: Alice spending her IOU twice (double spend), once transferring it to *B* and once to *P*.

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.

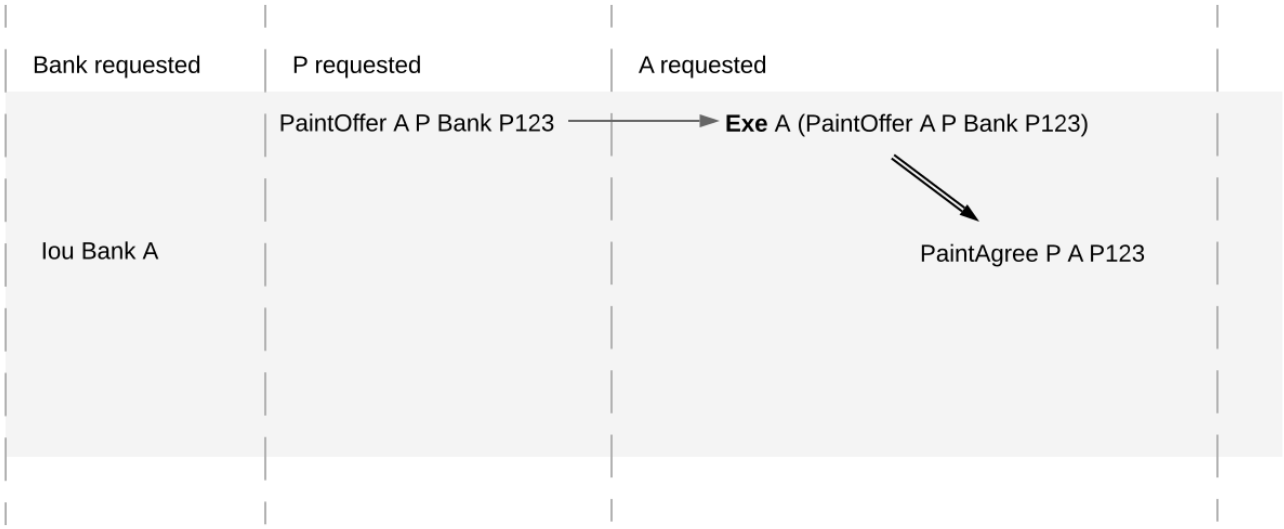


Fig. 2: Alice changing the offer's outcome by removing the transfer of the *Iou*.

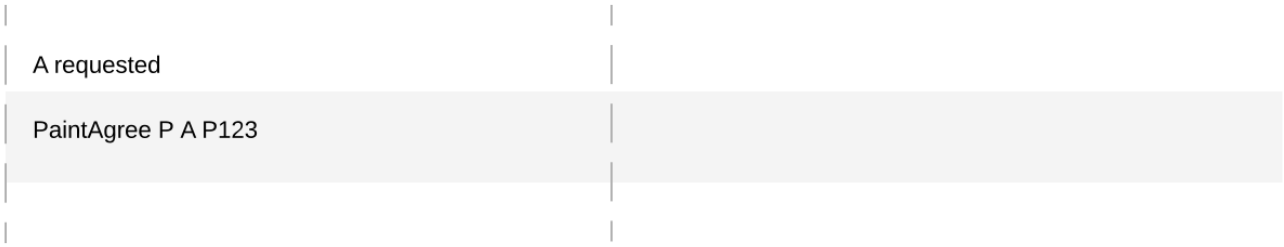


Fig. 3: An obligation imposed on the painter without his consent.

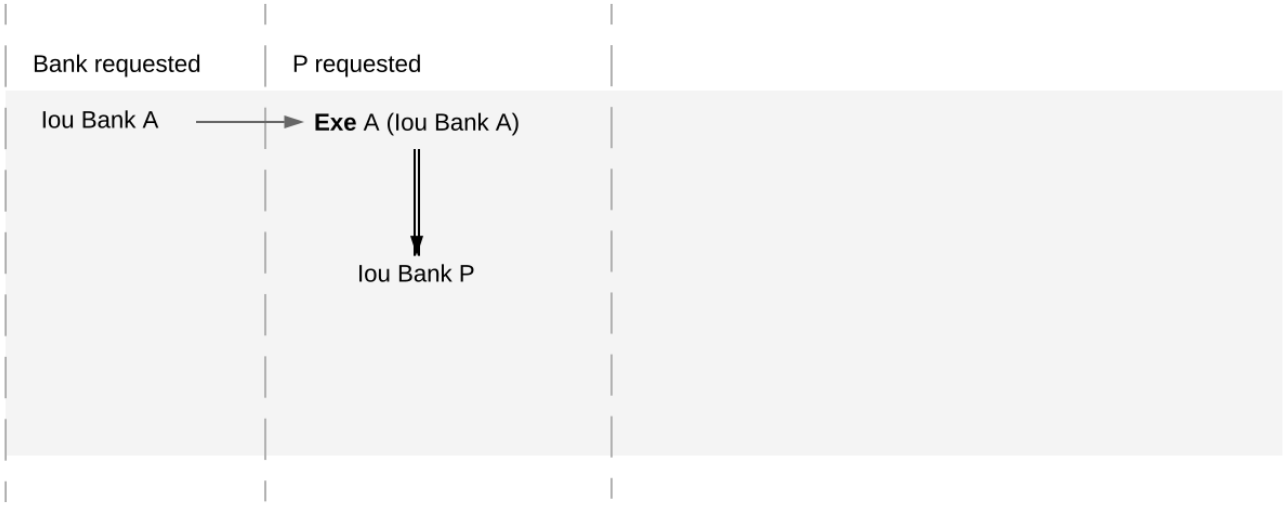


Fig. 4: Painter stealing Alice's IOU. Note that the ledger would be intuitively permissible if it was Alice performing the last commit.

P requested	P requested
PaintOffer A @P Bank &P123	**NoSuchKey** (P, P123)

Fig. 5: Painter falsely claiming that there is no offer.

P requested	P requested
PaintOffer A @P Bank &P123	PaintOffer David @P Bank &P123

Fig. 6: Painter trying to create two different paint offers with the same reference number.

5.2.2 Integrity

This section addresses the question of who can request which changes.

5.2.2.1 Valid Ledgers

At the core is the concept of a *valid ledger*; changes are permissible if adding the corresponding commit to the ledger results in a valid ledger. **Valid ledgers** are those that fulfill three conditions:

Consistency Exercises and fetches on inactive contracts are not allowed, i.e. contracts that have not yet been created or have already been consumed by an exercise. A contract with a contract key can be created only if the key is not associated to another unconsumed contract, and all key assertions hold.

Conformance Only a restricted set of actions is allowed on a given contract.

Authorization The parties who may request a particular change are restricted.

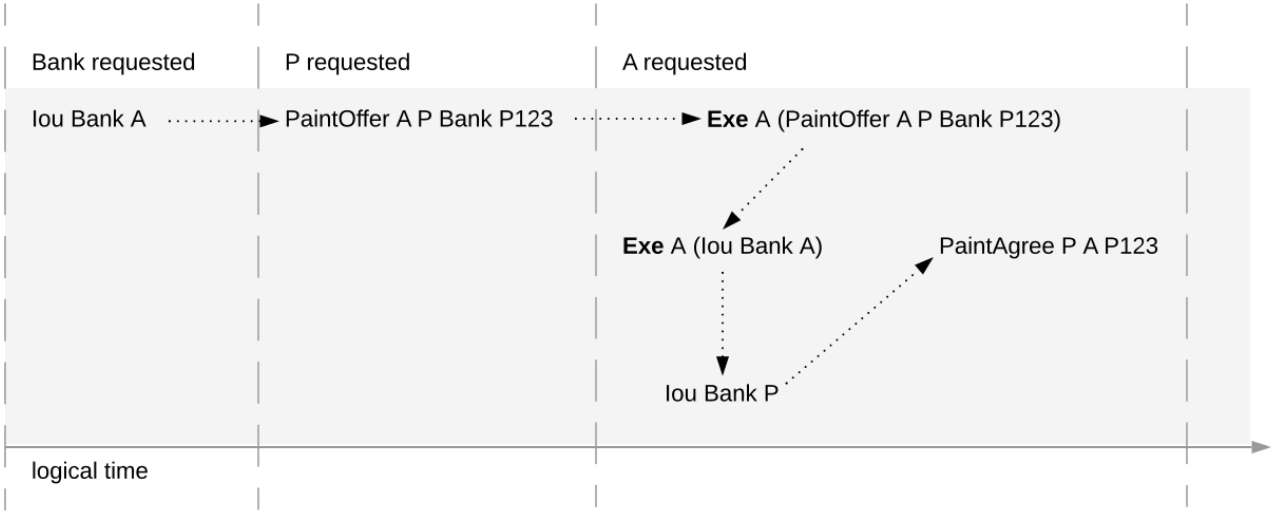
Only the last of these conditions depends on the party (or parties) requesting the change; the other two are general.

5.2.2.2 Consistency

Consistency consists of two parts:

1. *Contract consistency*: Contracts must be created before they are used, and they cannot be used once they are consumed.
2. *Key consistency*: Keys are unique and key assertions are satisfied.

To define this precisely, notions of before and after are needed. These are given by putting all actions in a sequence. Technically, the sequence is obtained by a pre-order traversal of the ledger's actions, noting that these actions form an (ordered) forest. Intuitively, it is obtained by always picking parent actions before their proper subactions, and otherwise always picking the actions on the left before the actions on the right. The image below depicts the resulting order on the paint offer example:



In the image, an action *act* happens before action *act'* if there is a (non-empty) path from *act* to *act'*. Then, *act'* happens after *act*.

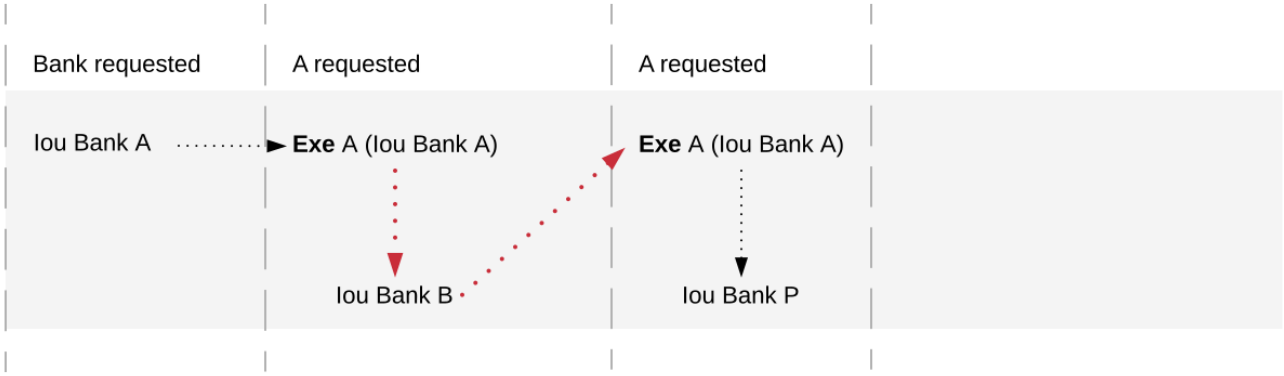
Contract consistency

Contract consistency ensures that contracts are used after they have been created and before they are consumed.

Definition contract consistency A ledger is **consistent for a contract c** if all of the following holds for all actions *act* on *c*:

1. either *act* is itself **Create c** or a **Create c** happens before *act*
2. *act* does not happen before any **Create c** action
3. *act* does not happen after any exercise 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.

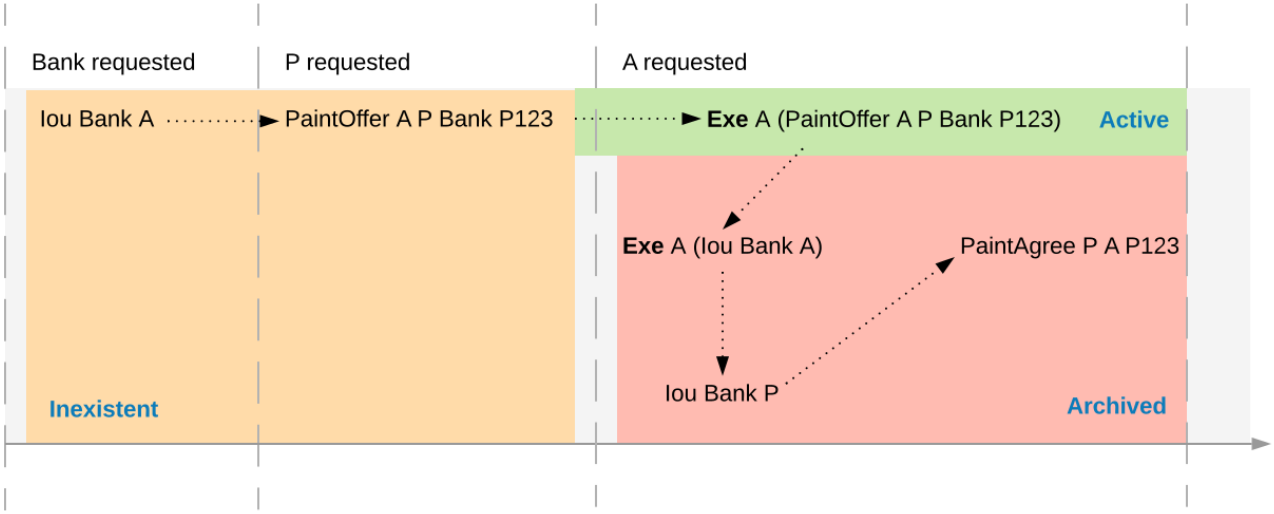


Fig. 7: Activeness of the *PaintOffer* contract

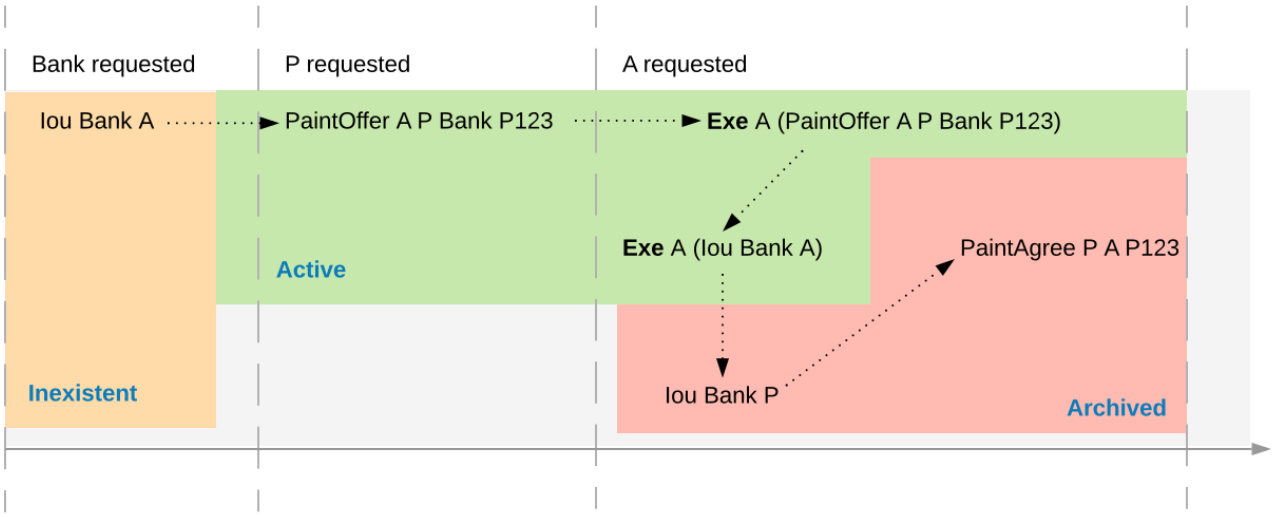


Fig. 8: Activeness of the *Iou Bank A* contract

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
> *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**.

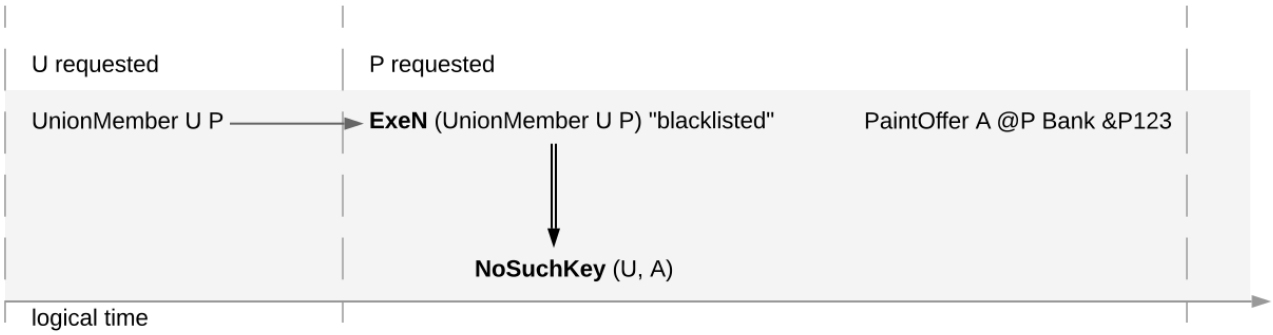
Key consistency ensures that there is at most one active contract for each key and that all key assertions are satisfies.

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 *C* 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*.



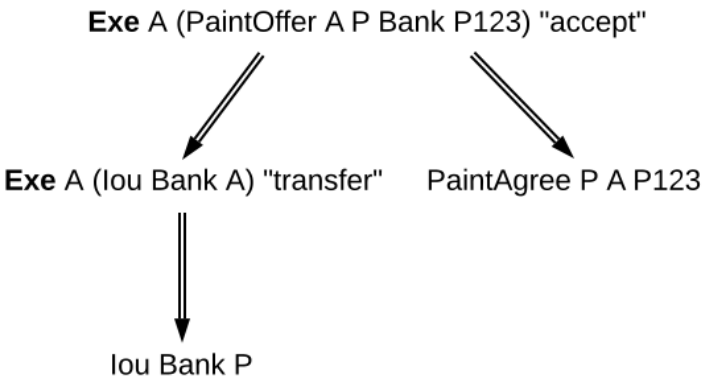
Key consistency extends to actions, transactions and lists of transactions just like the other consistency notions.

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 yield 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 transactions are the contracts that a transactions 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 no 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.

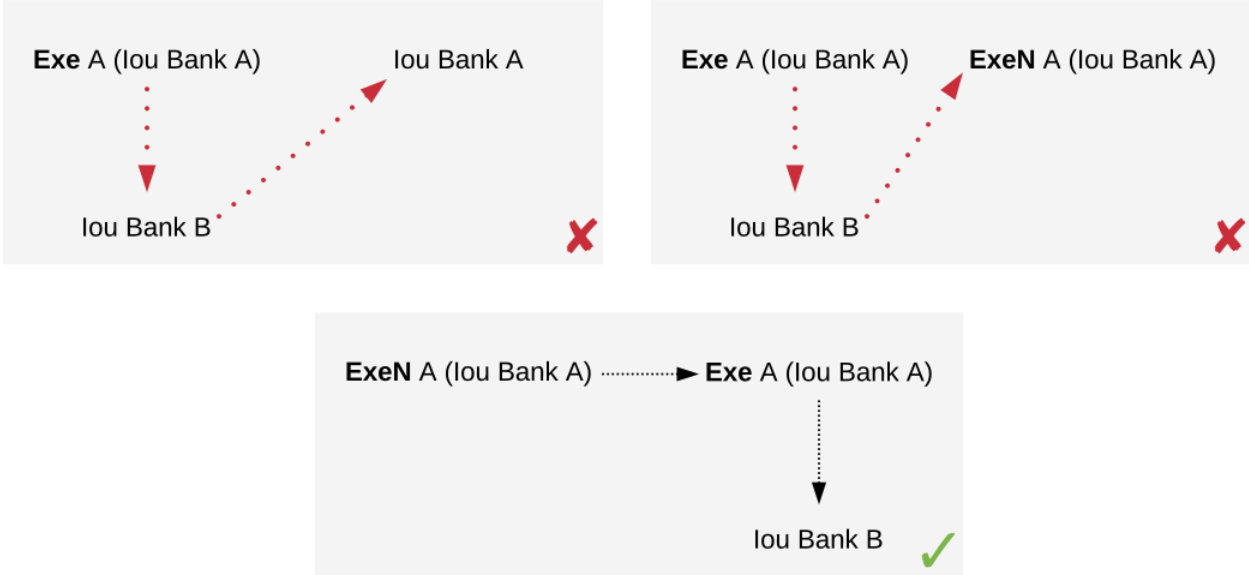


Fig. 9: The first two transactions violate the conditions of internal consistency. The first transaction creates the *Iou* after exercising it consumingly, violating both conditions. The second transaction contains a (non-consuming) exercise on the *Iou* after a consuming one, violating the second condition. The last transaction is internally consistent.

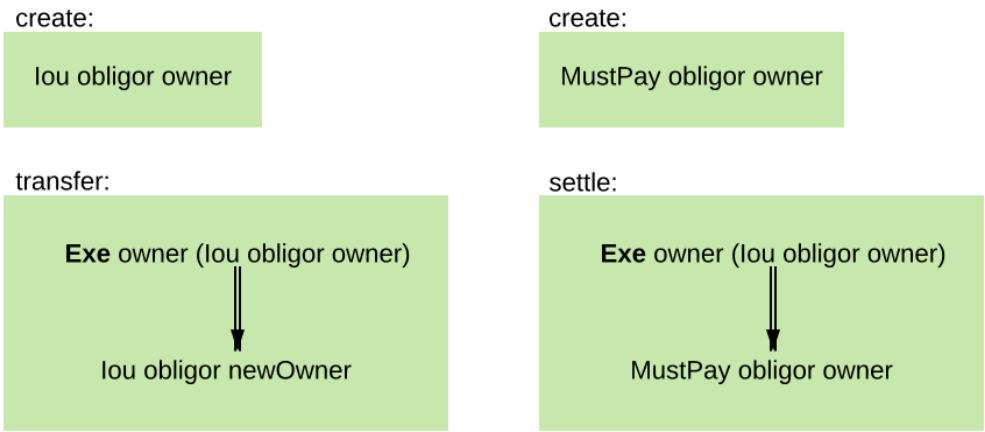
Definition input key A key *k* is an **input key** to an internally consistent transaction if the first action *act* on *k* is either a **Create** action or a **NoSuchKey** assertion.

In the *blacklisting example*, *P*'s transaction has two input keys: *(U, A)* due to the **NoSuchKey** action and *(P, P123)* as it creates a *PaintOffer* contract.

5.2.2.3 Conformance

The *conformance* condition constrains the actions that may occur on the ledger. This is done by considering a **contract model** *M* (or a **model** for short), which specifies the set of all possible actions. A ledger is **conformant to M** (or conforms to M) if all top-level actions on the ledger are members of *M*. Like consistency, the notion of conformance does not depend on the requesters of a commit, so it can also be applied to transactions and lists of transactions.

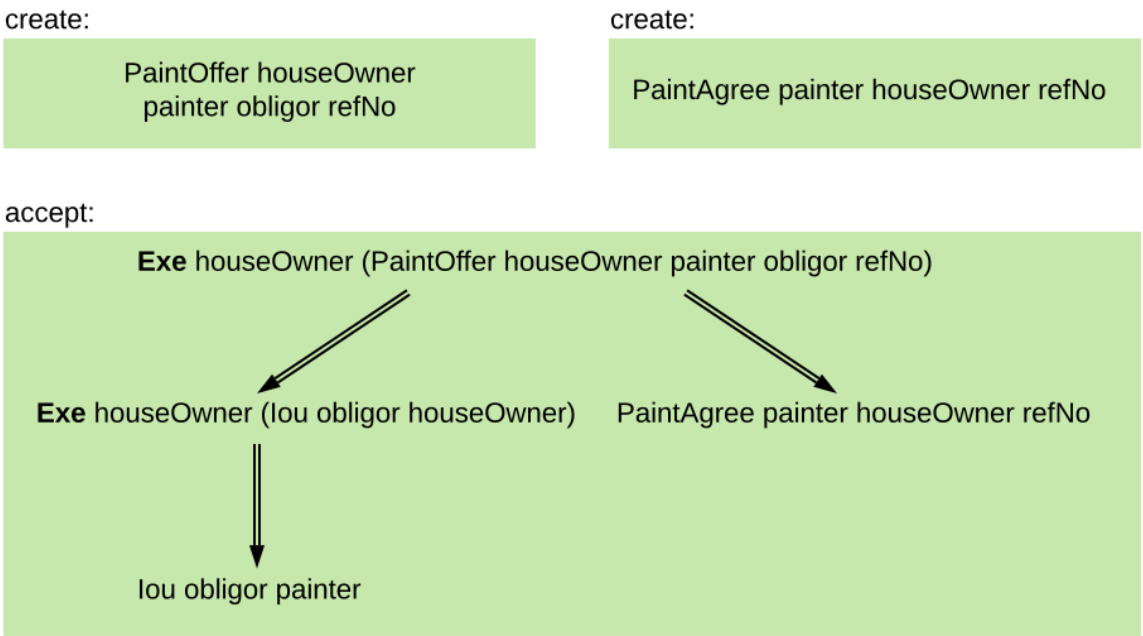
For example, the set of allowed actions on IOU contracts could be described as follows.



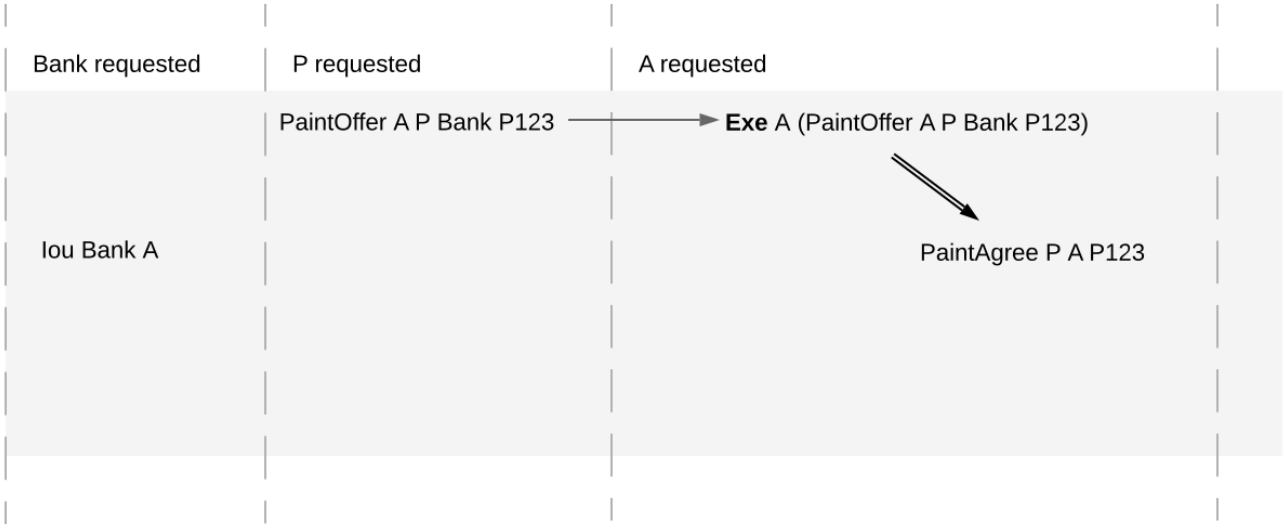
The boxes in the image are templates in the sense that the contract parameters in a box (such as obligor or owner) can be instantiated by arbitrary values of the appropriate type. To facilitate understanding, each box includes a label describing the intuitive purpose of the corresponding set of actions. As the image suggest, 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 arbitrary 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 *MustPaint* contracts.



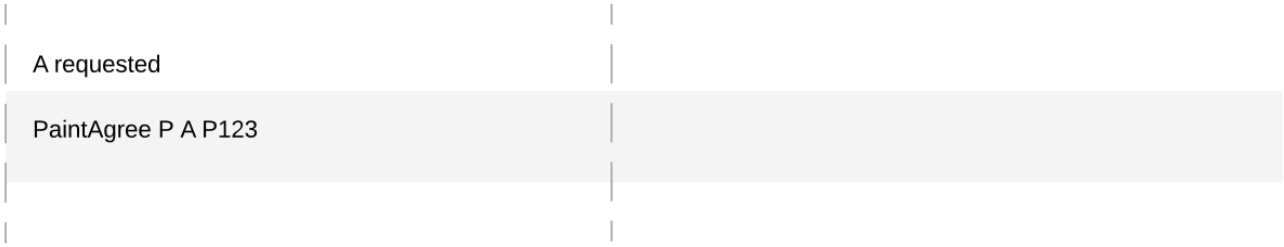
The problem with example where Alice changes the offer's outcome to avoid transferring the money now becomes apparent.



A's commit is not conformant to the contract model, as the model does not contain the top-level action she is trying to commit.

5.2.2.4 Authorization

The last criterion rules out the last two problematic examples, *an obligation imposed on a painter*, and *the painter stealing Alice's money*. The first of those is visualized below.



The reason why the example is intuitively impermissible is that the *PaintAgree* contract is supposed to express that the painter has an obligation to paint Alice's house, but he never agreed to that obligation. On paper contracts, obligations are expressed in the body of the contract, and imposed on the contract's *signatories*.

Signatories, Agreements, and Maintainers

To capture these elements of real-world contracts, the **contract model** additionally specifies, for each contract in the system:

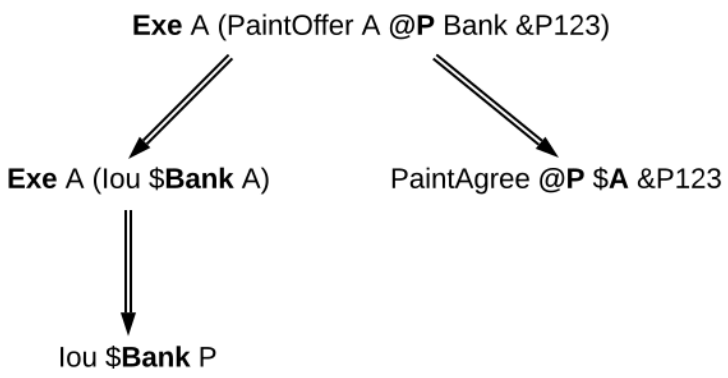
1. A non-empty set of **signatories**, the parties bound by the contract.
2. An optional **agreement text** associated with the contract, specifying the off-ledger, real-world obligations of the signatories.
3. If the contract is associated with a key, a non-empty set of **maintainers**, the parties that make sure that at most one unconsumed contract exists for the key. The maintainers must be a subset of the signatories and depend only on the key. This dependence is captured 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 they 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 DA ledger, a party can **authorize** a subaction of a commit in one of the two 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 DA 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:

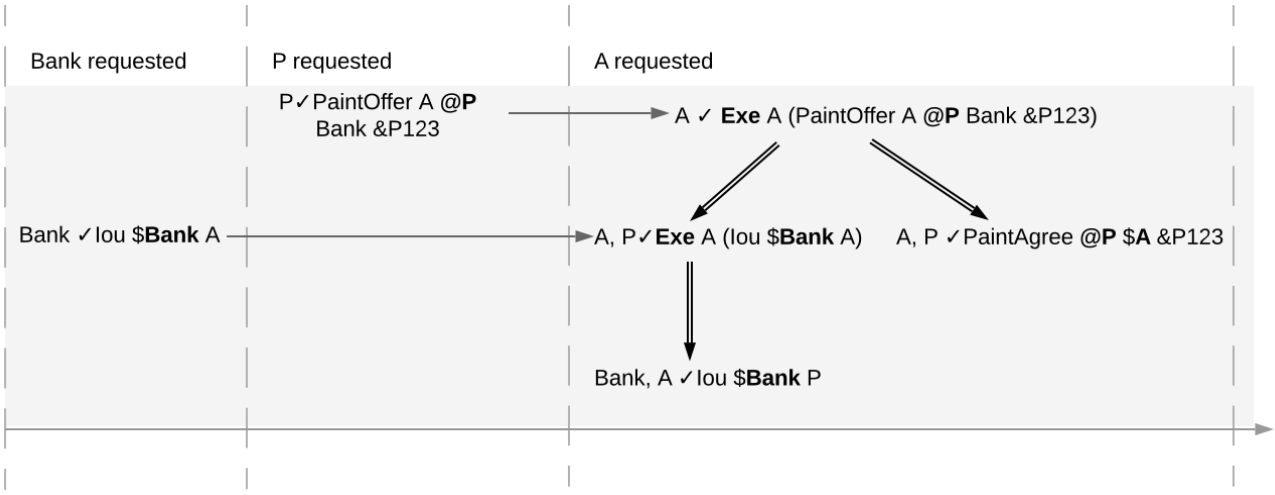
1. the required authorizers of a **Create** action on a contract *c* are the signatories of *c*.
2. the required authorizers of an **Exercise** or a **Fetch** action are its actors.
3. the required authorizers of a **NoSuchKey** assertion are the maintainers of the key.

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.

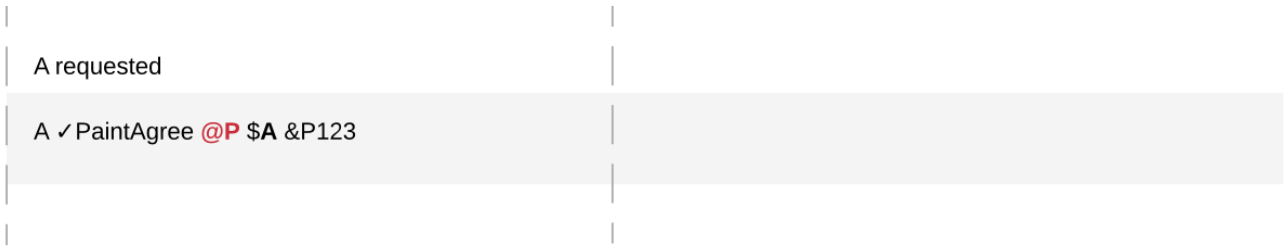
In the visualizations below, $\Pi \checkmark$ *act* denotes that the parties Π authorize the action *act*. The resulting authorizations are shown below.



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.

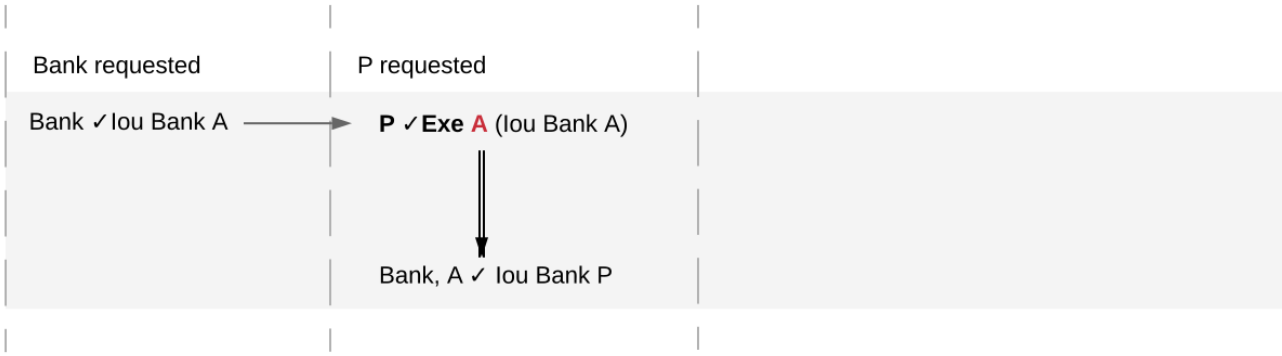
Similarly, the intuitively problematic examples are prohibitied by our authorization criterion. In the first example, Alice forced the painter to paint her house. The authorizations for the example are shown below.



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 as required authorizers for a **NoSuchKey** assertion is discussed in the next section about *privacy*.

5.2.2.5 Valid Ledgers, Obligations, Offers and Rights

DA 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.

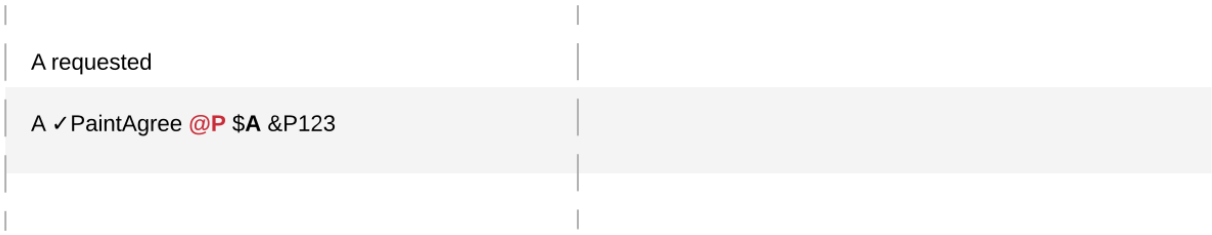
Second, every contract on a DA ledger can simultaneously model both:

> 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.

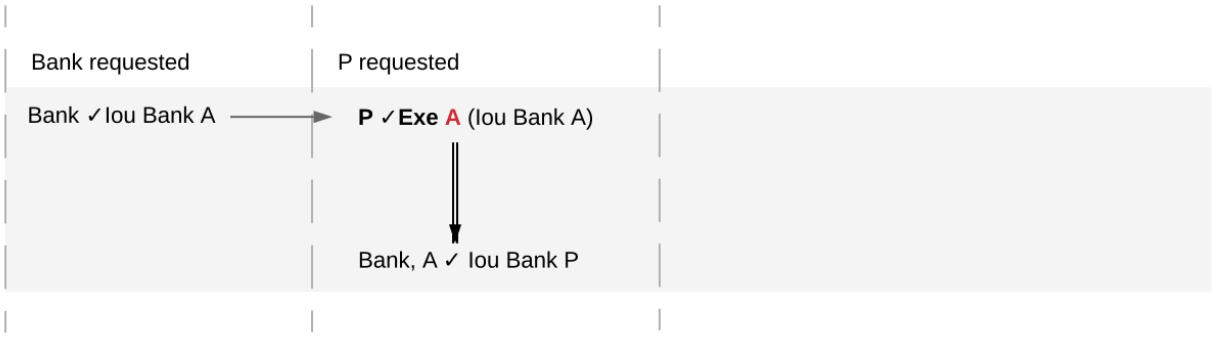
Third, in DA ledgers, as in the real world, one person's rights are another person's obligations. For example, *A*'s right to accept the *PaintOffer* is *P*'s obligation to paint her house in case she accepts. In DA ledgers, a party's rights according to a contract model are the exercise actions the party can perform according to the authorization and conformance rules.

Finally, validity conditions ensure three important properties of the DA ledger model, that mimic the contract law.

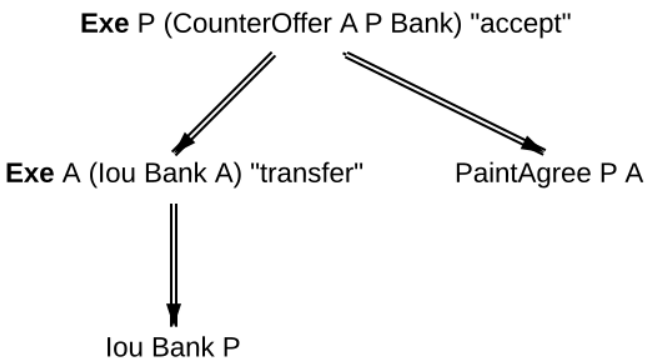
1. **Obligations need consent**. DA ledgers follow the offer-acceptance pattern of the contract law, and thus ensures that all ledger contracts are formed voluntarily. For example, the following ledger is not valid.



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 DA ledger, it can only be removed in accordance with the contract model. For example, assuming the IOU contract model shown earlier, if the ledger records the creation of a *MustPay* contract, the bank cannot later simply record an action that consumes this contract:

...	Bank commits
...	**Exe** Bank (MustPay Bank A)
MustPay Bank A	

That is, this ledger is invalid, as the action above is not conformant to the contract model.

5.2.3 Privacy

The previous sections have addressed two out of three questions posed in the introduction: what the ledger looks like, and who may request which changes. This section addresses the last one, who sees which changes and data. That is, it explains the privacy model for DA ledgers.

The privacy model of the DA platform is based on a **need-to-know basis**, and provides privacy **on the level of subtransactions**. Namely, a party learns only those parts of ledger changes that affect contracts in which the party has a stake, and the consequences of those changes. And maintainers see all changes to the contract keys they maintain.

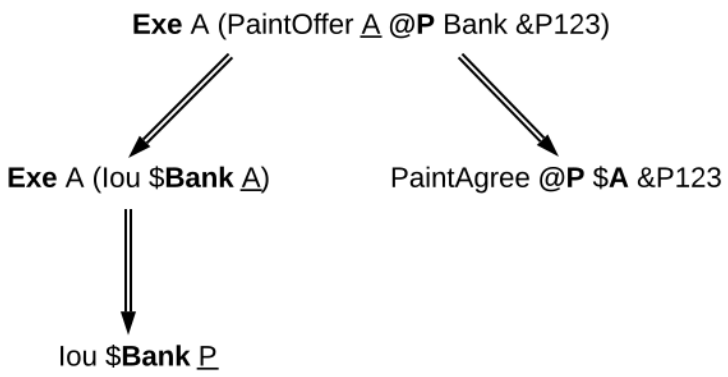
To make this more precise, a stakeholder concept is needed.

5.2.3.1 Contract Observers and Stakeholders

Intuitively, as signatories are bound by a contract, they have a stake in it. Actors might not be bound by the contract, but they still have a stake in their actions, as these are the actor's rights. Generalizing this, **observers** are parties who might not be bound by the contract, but still have the right to see the contract. For example, Alice should be an observer of the *PaintOffer*, such that she can 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 observers on each contract. A **stakeholder** of a contract (according to a given contract model) is then either a signatory or an observer on the contract. Note that in DAML, as detailed *later*, controllers specified using simple syntax are automatically made observers whenever possible.

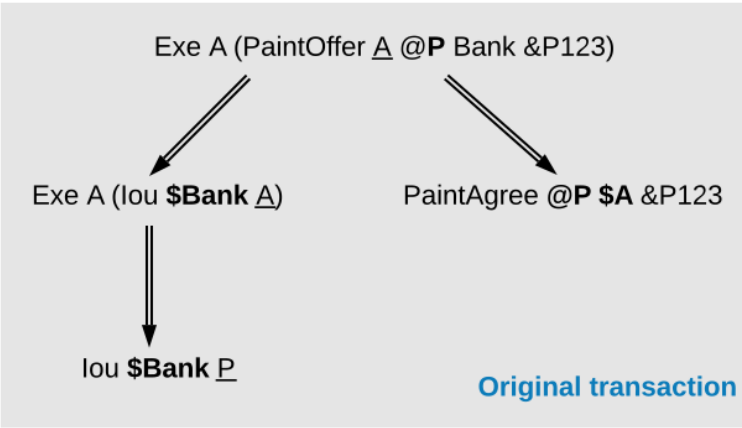
In the graphical representation of the paint offer acceptance below, observers who are not signatories are indicated by an underline.

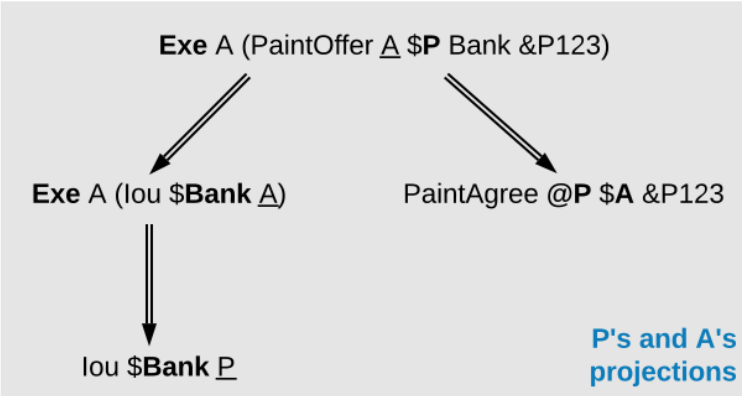


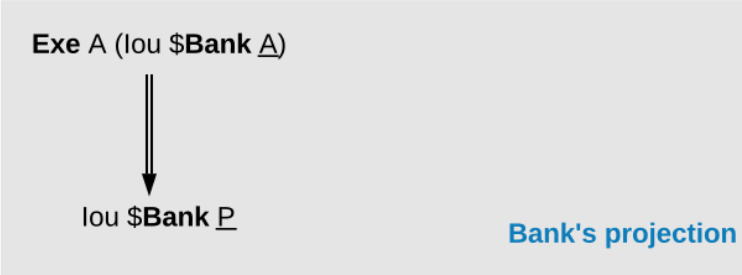
5.2.3.2 Projections

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 though *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, the DA Platform shows to observers on a contract only the *state changing* actions on the contract. More precisely, **Fetch** and non-consuming **Exercise** actions are not shown to the observers - except when they are the actors 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 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 $tx = act_1, , act_n$ for a party *p* is the sub-transaction obtained by doing the following for each action act_i:

1. If *p* is an informee of act_i, keep act_i as-is.
2. Else, if act_i has consequences, replace act_i by the projection (for *p*) of its consequences, which might be empty.
3. Else, drop act_i.

Finally, the **projection of a ledger** *l* for a party *p* is a list of transactions obtained by first projecting the transaction of each commit in *l* for *p*, and then removing all empty transactions from the result. Note that the projection of a ledger is not a ledger, but a list of transactions. Projecting the ledger of our complete paint offer example yields the following projections for each party:

Examine each party's projection in turn:

1. The painter does not see any part of the first commit, as he is not a stakeholder of the *Iou Bank A* contract. Thus, this transaction is not present in the projection for the painter at all. However, the painter is a stakeholder in the *PaintOffer*, so he sees both the creation and the exercise of this contract (again, recall that all consequences of an exercise action are a part of the action

itself).

2. Alice is a stakeholder in both the *Iou Bank A* and *PaintOffer A B Bank* contracts. As all top-level actions in the ledger are performed on one of these two contracts, Alice's projection includes all the transactions from the ledger intact.
3. The Bank is only a stakeholder of the IOU contracts. Thus, the bank sees the first commit's transaction as-is. The second commit's transaction is, however dropped from the bank's projection. The projection of the last commit's transaction is as described above.

Ledger projections do not always satisfy the definition of consistency, even if the ledger does. For example, in P's view, *Iou Bank A* is exercised without ever being created, and thus without being made active. Furthermore, projections can in general be non-conformant. However, the projection for a party *p* is always

> internally consistent for all contracts,
> consistent for all contracts on which *p* is a stakeholder, and
> consistent for the keys that *p* is a maintainer of.

In other words, *p* is never a stakeholder on any input contracts of its projection. Furthermore, if the contract model is **subaction-closed**, which means that for every action *act* in the model, all subactions of *act* are also in the model, then the projection is guaranteed to be conformant. As we will see shortly, DAML-based contract models are conformant. Lastly, as projections carry no information about the requesters, we cannot talk about authorization on the level of projections.

5.2.3.3 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, arbitrary other parties. Unlike **Create** actions to observers, such assertions are of no value to the other parties. Since processing such assertions may be expensive, they can be considered spam. Requiring all maintainers to authorize a **NoSuchKey** assertion avoids the problem.

5.2.3.4 Divulgence: When Non-Stakeholders See Contracts

The guiding principle for the privacy model of DA 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.

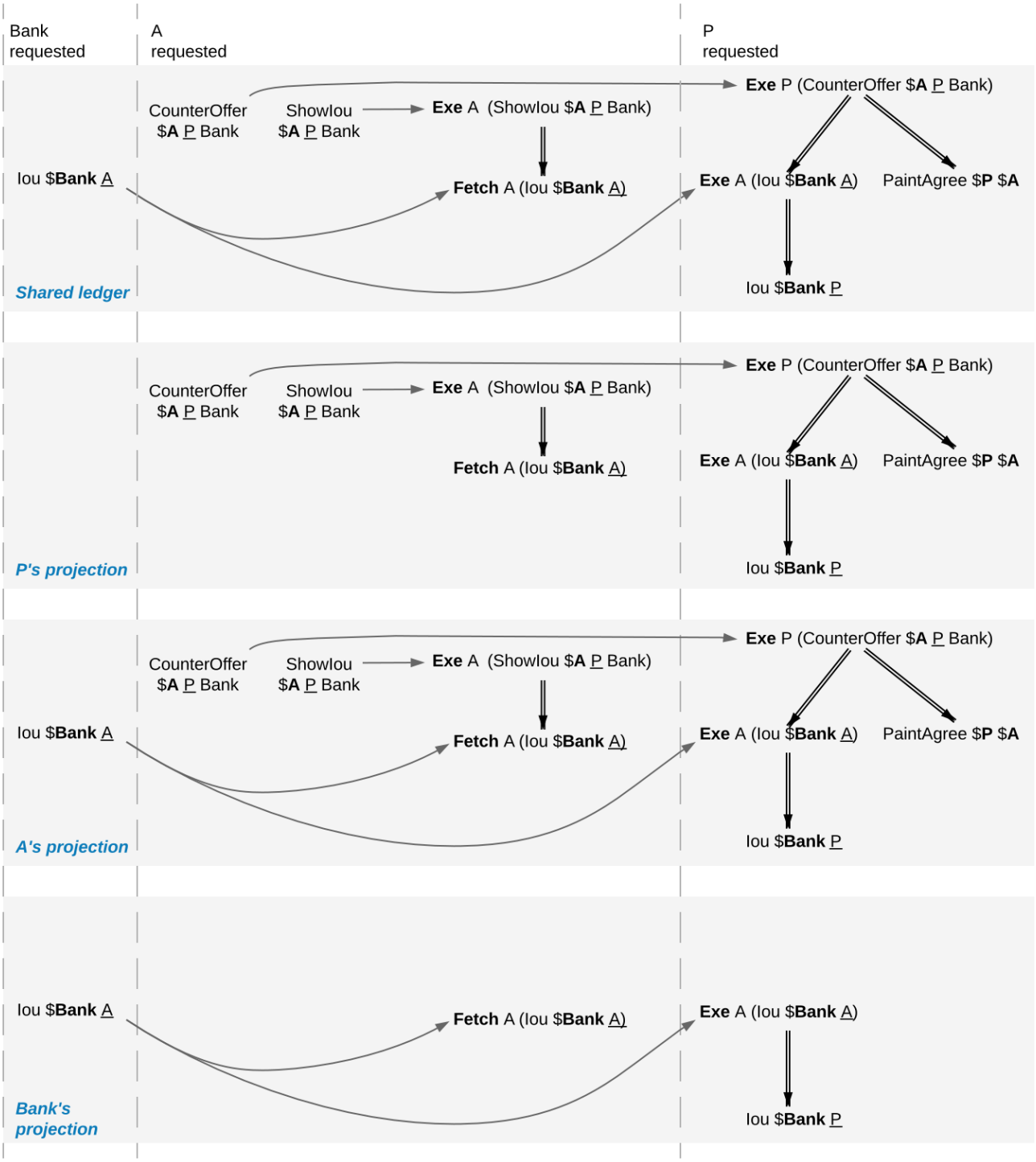
In the example of *ledger projections of the paint offer*, the exercise on the *PaintOffer* is visible both the painter and to Alice. As a consequence, the exercise on the *Iou Bank A* is visible to the painter, and the

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.

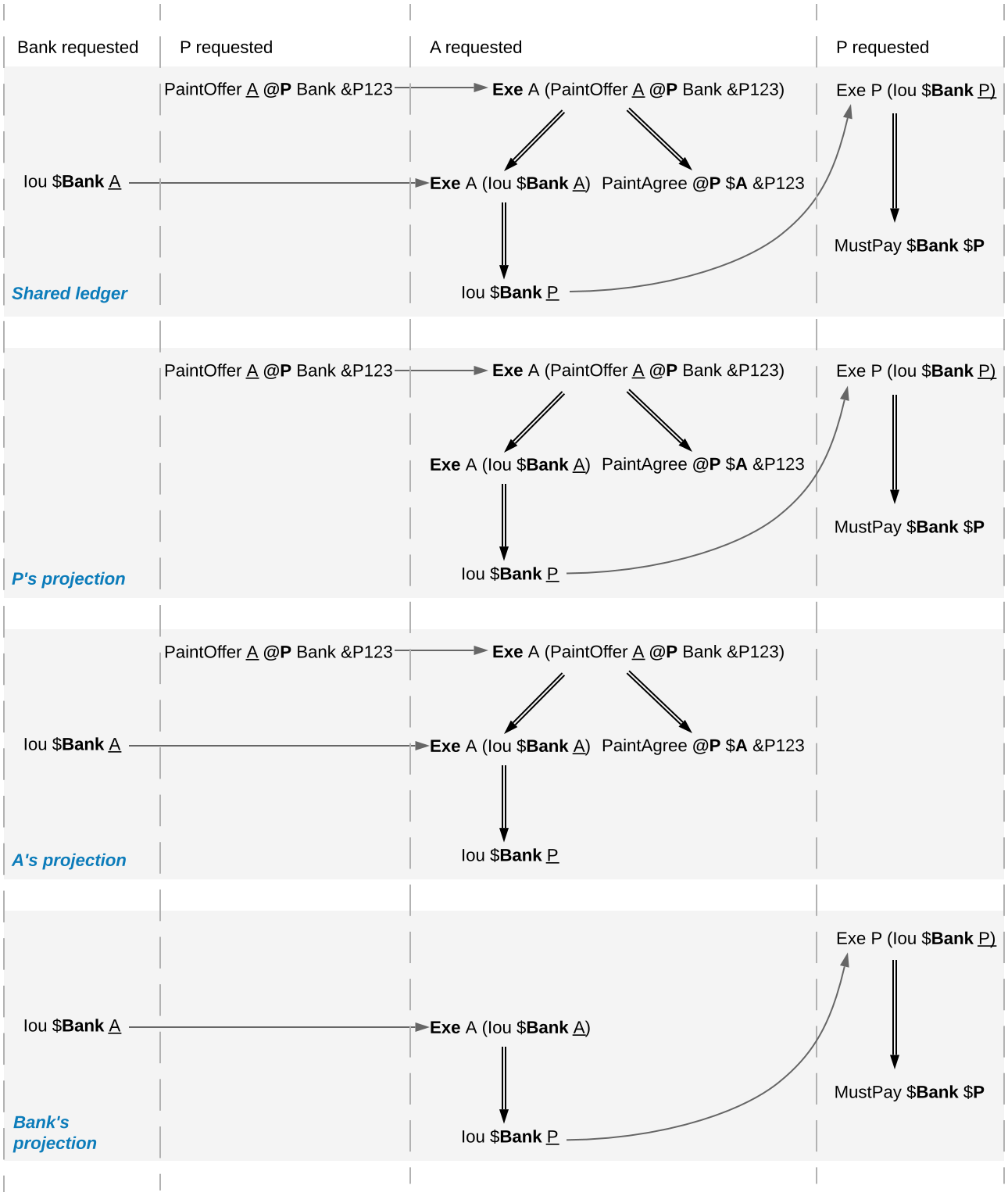
Showing contracts to non-stakeholders through ledger projections is called **divulgence**. Divulgence is a deliberate choice in the design of DA ledgers. In the paint offer example, the only proper way to accept the offer is to transfer the money from Alice to the painter. Conceptually, at the instant 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 an 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 know whether the contract is still active at any point after its creation. Similarly, in the previous example with the counteroffer, Alice could spend the IOU that she showed to the painter by the time the painter attempts to accept her counteroffer. In this case, the painter's transaction could not be added to the ledger, as it would result in a double spend and violate validity. But the painter has no way to predict whether his acceptance can be added to the ledger or not.

5.2.4 DAML: Defining Contract Models Compactly

As described in preceeding 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, 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 DAML templates such that the template arguments satisfy the *ensure* clause of the template
2. **Exercise** actions on a contract instance corresponding to DAML choices on that template, with given choice arguments, such that:
   1. The actors match the controllers of the choice. That is, the DAML controllers define the *required authorizers* of the choice.
   2. The exercise kind matches.
   3. All assertions in the update block hold for the given choice arguments.
   4. Create, exercise, fetch and key statements in the DAML 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 instance 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 signatories of *c2*.
   A *fetchByKey* statement also produce 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** or **contract instance**, 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 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 DAML 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 DAML contracts are created through a content-addressing scheme. This means every DAML 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 DAML templates. This model is subaction-closed; all exercise and create actions done within an update block are also always permissible as top-level actions.

Chapter 6

Deploying

6.1 Deploying to DAML ledgers

To run a DAML application, you'll need to deploy it to a DAML ledger.

6.1.1 How to deploy

You can deploy to:

The Sandbox with persistence. For information on how to do this, see the section on persistence in *DAML Sandbox* docs

Further deployment options which are in development. For information on these options and their stage of development, see the table below.

To deploy a DAML project to a ledger, you will need the ledger's hostname (or IP) and the port number for the gRPC Ledger API. The default port number is 6865. Then, inside your DAML project folder, run the following command, taking care to substitute the ledger's hostname and port for `<HOSTNAME>` and `<PORT>`:

```
$ daml deploy --host=<HOSTNAME> --port=<PORT>
```

This command will deploy your project to the ledger. This has two steps:

1. It will allocate the parties specified in the project's `daml.yaml` on the ledger if they are missing. The command looks through the list of parties known to the ledger, sees if any party is missing by comparing display names, and adds any missing party via the party management service of the Ledger API.
2. It will upload the project's compiled DAR file to the ledger via the package management service of the Ledger API. This will make the templates defined in the current project available to the users of the ledger.

Instead of passing `--host` and `--port` flags to the command above, you can add the following section to the project's `daml.yaml` file:

```yaml
ledger:
    host: <HOSTNAME>
    port: <PORT>
```

You can also use the `daml ledger` command for more fine-grained deployment options, and to interact with the ledger more generally. 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.

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
```

6.1.2 Available DAML Products

The following table lists commercially supported DAML ledgers and environments that are available for production use today.

Product	Ledger	Vendor
Sextant for DAML	Amazon Aurora	Blockchain Technology Partners
Sextant for DAML	Hyperledger Sawtooth	Blockchain Technology Partners
project : DABL	Managed cloud enviroment	Digital Asset

6.1.3 Open Source Integrations

The following table lists open source DAML integrations.

Ledger	Developer	More Information
Hyperledger Sawtooth	Blockchain Technology Partners	Github Repo
Hyperledger Fabric	Hacera	Github Repo
PostgreSQL	Digital Asset	DAML Sandbox Docs

6.1.4 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
Corda	R3	press release, June 2019
QLDB	Blockchain Technology Partners	press release, September 2019
Canton	Digital Asset reference implementation	canton.io

Chapter 7

Examples

7.1 DAML examples

We have plenty of example code, both of DAML and of applications around DAML, on our public GitHub organization.

- 12+ examples of different use cases: A repository containing a wide variety of DAML examples
- Bond trading example: DAML code and automation using the Java bindings
- Collateral management example: DAML code
- Repurchase agreement example: DAML code and automation using the Java bindings
- Upgrading DAML templates example: DAML code
- Java bindings tutorial: Three examples using the Java bindings with a very simple DAML model
- Node.js tutorial: Step-by-step running through using the experimental Node.js bindings

Chapter 8

Experimental features

8.1 WARNING

The tools described in this section are actively being designed and are subject to breaking changes or removal. When we become more confident in their designs, we will introduce them as standard components in the SDK.

8.1.1 Navigator Console

8.1.1.1 Querying the Navigator local database

You can query contracts, transactions, events, or commands in any way you'd like, by querying the Navigator Console's local database(s) directly. This page explains how you can run queries.

Note: Because of the strong DAML privacy model, each party will see a different subset of the ledger data. For this reason, each party has its own local database.

The Navigator database is implemented on top of SQLite. SQLite understands most of the standard SQL language. For information on how to compose SELECT statements, see to the SQLite SELECT syntax specification.

To run queries, use the `sql` Navigator Console command. Take a look at the examples below to see how you might use this command.

On this page:

How the data is structured
Example query using plain SQL
Example queries using JSON functions

How the data is structured

To get full details of the schema, run `sql_schema`.

Semi-structured data (such as contract arguments or template parameters) are stored in columns of type JSON.

You can compose queries against the content of JSON columns by using the SQLite functions json_extract and json_tree.

Example query using plain SQL

Filter on the template id of contracts:

```
sql select count (*) from contract where template_id like '%Offer%'
```

Example queries using JSON functions

Select JSON fields from a JSON column by specifying the path:

```
sql select json_extract(value, '$.argument.landlord') from contract
```

Filter on the value of a JSON field:

```
sql select contract.id, json_tree.fullkey  from contract, json_
↪tree(contract.value) where atom is not null and json_tree.value like '
↪%BANK1%'
```

Filter on the JSON key and value:

```
sql select contract.id from contract, json_tree(contract.value) where atom
↪is not null and json_tree.key = 'landlord' and json_tree.value like '
↪%BANK1%'
```

Filter on the value of a JSON field for a given path:

```
sql select contract.id from contract where json_extract(contract.value, '$.
↪argument.landlord') like '%BANK1%'
```

Identical query using json_tree:

```
sql select contract.id from contract, json_tree(contract.value) where atom
↪is not null and json_tree.fullkey = '$.argument.landlord' and json_tree.
↪value like '%BANK1%'
```

Filter on the content of an array if the index is specified:

```
sql select contract.id from contract where json_extract(contract.value, '$.
↪template.choices[0].name') = 'Accept'
```

Filter on the content of an array if the index is not specified:

```
sql select contract.id from contract, json_tree(contract.value) where atom
↪is not null and json_tree.path like '$.template.choices[%]' and json_
↪tree.value ='Accept'
```

The Navigator Console is a terminal-based front-end for inspecting and modifying a Digital Asset ledger. It's useful for DAML developers, app developers, or business analysts who want to debug or analyse a ledger by exploring it manually.

You can use the Console to:

inspect available templates
query active contracts
exercise commands
list blocks and transactions

If you prefer to use a graphical user interface for these tasks, use the *Navigator* instead.

On this page:

Try out the Navigator Console on the Quickstart
- *Installing and starting Navigator Console*
- *Getting help*
- *Exiting Navigator Console*
- *Using commands*

Displaying status information
Choosing a party
Advancing time
Inspecting templates
Inspecting contracts, transactions, and events
Querying data
Creating contracts
Exercising choices
- *Advanced usage*

Using Navigator outside the SDK
Using Navigator with the Digital Asset ledger

8.1.1.2 Try out the Navigator Console on the Quickstart

With the sandbox running the *quickstart application*

1. To start the shell, run `daml navigator console localhost 6865`
   This connects Navigator Console to the sandbox, which is still running.
   You should see a prompt like this:

```
  _  __          _           __
 / |/ /__ __  __(_)__ ____ _/ /____  ____
/    / _ `/ |/ / / _ `/ _ `/ __/ _ \/ __/
/_/|_/\_,_/|___/_/\_, /\_,_/\__/\___/_/
                 /___/
Version 1.1.0
Welcome to the console. Type 'help' to see a list of commands.
```

2. Since you are connected to the sandbox, you can be any party you like. Switch to Bob by running:
   `party Bob`
   The prompt should change to `Bob>`.
3. Issue a *BobsCoin* to yourself. Start by writing the following, then hit Tab to auto-complete and get the full name of the *Iou.Iou* template:
   `create Iou.Iou <TAB>`
   This full name includes a hash of the DAML package, so don't copy it from the command below - it's better to get it from the auto-complete feature.
   You can then create the contract by running:
   `create Iou.Iou@317057d06d4bc4bb91bf3cfe3292bf3c2467c5e004290e0ba20b993eb1e40931`

`with {issuer="Bob", owner="Bob", currency="BobsCoin", amount="1.0", observers=[]}`
You should see the following output:

```
CommandId: 1b8af77a91ad1102
Status: Success
TransactionId: 10
```

4. You can see details of that contract using the TransactionId. First, run:
`transaction 10`
to get details of the transaction that created the contract:

```
Offset: 11
Effective at: 1970-01-01T00:00:00Z
Command ID: 1b8af77a91ad1102
Events:
- [#10:0] Created #10:0 as Iou
```

Then, run:
`contract #10:0`
to see the contract for the new BobsCoin:

```
Id: #10:0
TemplateId: Iou.
↪Iou@317057d06d4bc4bb91bf3cfe3292bf3c2467c5e004290e0ba20b993eb1e40931
Argument:
  observers:

  issuer: Bob
  amount: 1.0
  currency: BobsCoin
  owner: Bob
Created:
  EventId: #10:0
  TransactionId: 10
  WorkflowId: 1ba8521c395096e3
Archived: Contract is active
```

5. You can transfer the coin to Alice by running:
`exercise #10:0 Iou_Transfer with {newOwner="Alice"}`

There are lots of other things you can do with the Navigator Console.

One of its most powerful features is that you can query its local databases using SQL, with the `sql` command.
For example, you could see all of the *Iou* contracts by running `sql select * from contract where template_id like 'Iou.Iou@%'`. For more examples, take a look at the *Navigator Console database documentation*.
For a full list of commands, run `help`. You can also look at the *Navigator Console documentation page*.
For help on a particular command, run `help <name of command>`.
When you are done exploring the shell, run `quit` to exit.

Installing and starting Navigator Console

Navigator Console is installed as part of the DAML SDK. See *Installing the SDK* for instructions on how to install the DAML SDK.

If you want to use Navigator Console independent of the SDK, see the *Advanced usage* section.

To run Navigator Console:

1. Open a terminal window and navigate to your DAML SDK project folder.
2. If the Sandbox isn't already running, run it with the command `daml start`.
   The sandbox prints out the port on which it is running - by default, port `6865`.
3. Run `daml navigator console localhost 6865`. Replace `6865` by the port reported by the sandbox, if necessary.

When Navigator Console starts, it displays a welcome message:

```
   _  __           _            __
  / |/ /__ __  __(_)__ ____ _/ /____  ____
 /    / _ `/ |/ / / _ `/ _ `/ __/ _ \/ __/
/_/|_/\_,_/|___/_/\_, /\_,_/\__/\___/_/
                  /___/
Version X.Y.Z
Welcome to the console. Type 'help' to see a list of commands.
```

Getting help

To see all available Navigator Console commands and how to use them, use the `help` command:

```
>help
Available commands:
choice                Print choice details
command               Print command details
commands              List submitted commands
contract              Print contract details
create                Create a contract
diff_contracts        Print diff of two contracts
event                 Print event details
exercise              Exercises a choice
help                  Print help
graphql               Execute a GraphQL query
graphql_examples      Print some example GraphQL queries
graphql_schema        Print the GraphQL schema
info                  Print debug information
package               Print DAML-LF package details
packages              List all DAML-LF packages
parties               List all parties available in Navigator
party                 Set the current party
quit                  Quit the application
set_time              Set the (static) ledger effective time
templates             List all templates
template              Print template details
time                  Print the ledger effective time
```

(continues on next page)

(continued from previous page)

```
transaction          Print transaction details
version              Print application version
sql_schema           Return the database schema
sql                  Execute a SQL query
```

To see the help for the given command, run `help <command>`:

```
>help create
Usage: create <template> with <argument>

Create a contract
Parameters:
<template>           Template ID
<argument>           Contract argument
```

Exiting Navigator Console

To exit Navigator Console, use the `quit` command:

```
>quit
Bye.
```

Using commands

This section describes how to use some common commands.

Note: Navigator Console has several features to help with typing commands:

Press the **Tab** key one or more times to use auto-complete and see a list of suggested text to complete the command.
Press the **Up** or **Down** key to scroll through the history of recently used commands.
Press **Ctrl+R** to search in the history of recently used commands.

8.1.1.3 Displaying status information

To see useful information about the status of both Navigator Console and the ledger, use the `info` command:

```
>info
   _  __           _            __
  / |/ /__ __   __(_)__ ____ _/ /____  ____
 /    / _ `/ |/ / / _ `/ _ `/ __/ _ \/ __/
/_/|_/\_,_/|___/_/\_, /\_,_/\__/\___/_/
                   /___/
Version 1.0.14 commit a3e1d1c30d84261fa9b6db95c69036da14bc9e7e
General info:
    Ledger host: localhost
    Ledger port: 6865
```

(continues on next page)

(continued from previous page)

```
    Secure connection: false
    Application ID: Navigator-c06fae89-d8ed-4656-b085-388e24569ecf
↪#5b21103194967935
Ledger info:
    Connection status: Connected
    Ledger ID: sandbox-051e2468-c679-43df-b99f-9c72dcd8ffa0
    Ledger time: 1970-01-01T00:16:40Z
    Ledger time type: static
Akka system:
    OPERATOR: Actor running
    BANK2: Actor running
    BANK1: Actor running
Local data:
    BANK1:
        Packages: 1
        Contracts: 0
        Active contracts: 0
        Last transaction: ???
    BANK2:
        Packages: 1
        Contracts: 0
        Active contracts: 0
        Last transaction: ???
    OPERATOR:
        Packages: 1
        Contracts: 1001
        Active contracts: 1001
        Last transaction: scenario-transaction-2002
```

8.1.1.4 Choosing a party

Privacy is an important aspect of a Digital Asset ledger: parties can only access the contracts on the ledger that they are authorized to. This means that, before you can interact with the ledger, you must assume the role of a particular party.

The currently active party is displayed left of the prompt sign (>). To assume the role of a different party, use the `party` command:

```
BANK1>party BANK2
BANK2>
```

Note: The list of available parties is configured when the Sandbox starts. (See the *DAML Assistant (daml)* or *Advanced usage* for more instructions.)

8.1.1.5 Advancing time

You can advance the time of the DAML Sandbox. This can be useful when testing, for example, when entering a trade on one date and settling it on a later date.

(For obvious reasons, this feature does not exist on the Digital Asset ledger.)

To display the current ledger time, use the `time` command:

```
>time
1970-01-01T00:16:40Z
```

To advance the time to the time you specify, use the `set_time` command:

```
>set_time 1970-01-02T00:16:40Z
New ledger effective time: 1970-01-02T00:16:40Z
```

8.1.1.6 Inspecting templates

To see what templates are available on the ledger you are connected to, use the `templates` command:

```
>templates
╔═══════════════════════╦═══════╦═══════╗
║Name                   ╒Package ╒Choices╕
╠═══════════════════════╬═══════╬═══════╣
║Main.RightOfUseAgreement╒07ca8611╒0      ╕
║Main.RightOfUseOffer   ╒07ca8611╒1      ╕
╚═══════════════════════╩═══════╩═══════╝
```

To get detailed information about a particular template, use the `template` command:

```
>template Offer<Tab>
>template Main.
↪RightOfUseOffer@07ca8611d05ec14ea4b973192ef6caa5d53323bba50720a8d7142c2a246cfb73
Name: Main.RightOfUseOffer
Parameter:
    landlord: Party
    tenant: Party
    address: Text
    expirationDate: Time
Choices:
- Accept
```

Note: Remember to use the **Tab** key. In the above example, typing Offer followed by the **Tab** key auto-completes the fully qualified name of the RightOfUseOffer template.

To get detailed information about a choice defined by a template, use the `choice` command:

```
>choice Main.RightOfUseOffer Accept
Name: Accept
Consuming: true
Parameter: Unit
```

8.1.1.7 Inspecting contracts, transactions, and events

The ledger is a record of transactions between authorized participants on the distributed network. Transactions consist of events that create or archive contracts, or exercise choices on them.

To get detailed information about a ledger object, use the singular form of the command (`transaction`, `event`, `contract`):

```
>transaction 2003
Offset: 1004
Effective at: 1970-01-01T00:16:40Z
Command ID: 732f6ac4a63c9802
Events:
- [#2003:0] Created #2003:0 as RightOfUseOffer
```

```
>event #2003:0
Id: #2003:0
ParentId: ???
TransactionId: 2003
WorkflowId: e13067beec13cf4c
Witnesses:
- Scrooge_McDuck
Type: Created
Contract: #2003:0
Template: Main.RightOfUseOffer
Argument:
    landlord: Scrooge_McDuck
    tenant: Bentina_Beakley
    address: McDuck Manor, Duckburg
    expirationDate: 2020-01-01T00:00:00Z
```

```
>contract #2003:0
Id: #2003:0
TemplateId: Main.RightOfUseOffer
Argument:
    landlord: Scrooge_McDuck
    tenant: Bentina_Beakley
    address: McDuck Manor, Duckburg
    expirationDate: 2020-01-01T00:00:00Z
Created:
    EventId: #2003:0
    TransactionId: 2003
    WorkflowId: e13067beec13cf4c
Archived: Contract is active
Exercise events:
```

8.1.1.8 Querying data

To query contracts, transactions, events, or commands in any way you'd like, you can query the Navigator Console's local database(s) directly.

Because of the strong DAML privacy model, each party will see a different subset of the ledger data. For this reason, each party has its own local database.

To execute a SQL query against the local database for the currently active party, use the `sql` command:

```
>sql select id, template_id, archive_transaction_id from contract
╔═══════╤═══════════════════╤═══════════════════════╗
╟id     │template_id        │archive_transaction_id╟
╟───────┼───────────────────┼───────────────────────╟
╟#2003:0│Main.RightOfUseOffer│null                   ╟
╟#2004:0│Main.RightOfUseOffer│null                   ╟
╚═══════╧═══════════════════╧═══════════════════════╝
```

See the *Navigator Local Database* documentation for details on the database schema and how to write SQL queries.

Note: The local database contains a copy of the ledger data, created using the Ledger API. If you modify the local database, you might break Navigator Console, but it will not affect the data on the ledger in any way.

8.1.1.9 Creating contracts

Contracts in a ledger can be created directly from a template, or when you exercise a choice. You can do both of these things using Navigator Console.

To create a contract of a given template, use the `create` command. The contract argument is written in JSON format (DAML primitives are strings, DAML records are objects, DAML lists are arrays):

```
>create Main.
↪RightOfUseOffer@07ca8611d05ec14ea4b973192ef6caa5d53323bba50720a8d7142c2a246cfb73
↪with {"landlord": "BANK1", "tenant": "BANK2", "address": "Example Street
↪", "expirationDate": "2018-01-01T00:00:00Z"}
CommandId: 1e4c1610eadba6b
Status: Success
TransactionId: 2005
```

Note: Again, you can use the **Tab** key to auto-complete the template name.

The Console waits briefly for the completion of the create command and prints basic information about its status. To get detailed information about your create command, use the `command` command:

```
>command 1e4c1610eadba6b
Command:
    Id: 1e4c1610eadba6b
    WorkflowId: a31ea1ca20cd5971
    PlatformTime: 1970-01-02T00:16:40Z
    Command: Create contract
    Template: Main.RightOfUseOffer
    Argument:
        landlord: Scrooge_McDuck
        tenant: Bentina_Beakley
        address: McDuck Manor, Duckburg
```

(continues on next page)

(continued from previous page)

```
        expirationDate: 2020-01-01T00:00:00Z
Status:
    Status: Success
    TransactionId: 2005
```

8.1.1.10 Exercising choices

To exercise a choice on a contract with the given ID, use the `exercise` command:

```
>exercise #2005:0 Accept
CommandID: 8dbbcbc917c7beee
Status: Success
TransactionId: 2006
```

```
>exercise #2005:0 Accept with {tenant="BANK2"}
CommandID: 8dbbcbc917c7beee
Status: Success
TransactionId: 2006
```

Advanced usage

8.1.1.11 Using Navigator outside the SDK

This section explains how to work with the Navigator if you have a project created outside of the normal SDK workflow and want to use the Navigator to inspect the ledger and interact with it.

Note: If you are using the Navigator as part of the DAML SDK, you do not need to read this section.

The Navigator is released as a fat Java *.jar* file that bundles all required dependencies. This JAR is part of the SDK release and can be found using the SDK Assistant's `path` command:

```
da path navigator
```

To launch the Navigator JAR and print usage instructions:

```
da run navigator
```

Provide arguments at the end of a command, following a double dash. For example:

```
da run navigator -- console \
  --config-file my-config.conf \
  --port 8000 \
  localhost 6865
```

The Navigator needs a configuration file specifying each user and the party they act as. It has a `.conf` ending by convention. The file has this format:

```
users {
    <USERNAME> {
```

(continues on next page)

(continued from previous page)

```
        party = <PARTYNAME>
    }
    ..
}
```

In many cases, a simple one-to-one correspondence between users and their respective parties is sufficient to configure the Navigator. Example:

```
users {
    BANK1 { party = "BANK1" }
    BANK2 { party = "BANK2" }
    OPERATOR { party = "OPERATOR" }
}
```

8.1.1.12 Using Navigator with the Digital Asset ledger

By default, Navigator is configured to use an unencrypted connection to the ledger.

To run Navigator against a secured Digital Asset 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
```

8.1.2 Extractor

8.1.2.1 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)

8.1.2.2 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
```

8.1.2.3 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.

8.1.2.4 Running the Extractor

The basic command to run the Extractor is:

```
$ daml extractor [options]
```

For what options to use, see the next sections.

8.1.2.5 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.

8.1.2.6 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]
```

8.1.2.7 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
  -p, --ledger-port <p>    The port of the Ledger host. Default is 6865.
  --party <value>          The party whose contract data should be
↪extracted.
  --from <value>           The transaction offset (exclusive) for the
↪snapshot start position.
                           Must not be greater than the current latest
↪transaction offset.
```

(continues on next page)

(continued from previous page)

```
                              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 the trusted
↪root CA.
```

Some options are tied to a specific subcommand, like `--connecturl` only makes sense for the `postgresql`, while others are general, like `--party`.

8.1.2.8 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.

8.1.2.9 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()
```

(continues on next page)

(continued from previous page)

```
  ,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 the DAML Ledger, 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`.

8.1.2.10 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
  ,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*.

8.1.2.11 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.

8.1.2.12 JSON format

Values on the ledger can be either primitive types, user-defined `records`, or `variants`. An extracted contract is represented in the database as a `record` of its create argument, and the fields of that `records` are either primitive types, other `records`, or `variants`. A contract can be a recursive structure of arbitrary depth.

These types are translated to JSON types the following way:

Primitive types

`ContractID`: represented as string.
`Int64`: represented as string.
`Decimal`: A decimal value with precision 38 (38 decimal digits), of which 10 after the comma / period. Represented as string.
`List`: represented as array.
`Text`: represented as string.
`Date`: days since the unix epoch. represented as integer.
`Time`: Microseconds since the UNIX epoch. Represented as number.
`Bool`: represented as boolean.

`Party`: represented as string.
`Unit` and `Empty` are represented as empty records.
`Optional`: represented as object, as it was a `Variant` with two possible constructors: `None` and `Some`.

User-defined types

`Record`: represented as object, where each create parameter's name is a key, and the parameter's value is the JSON-encoded value.
`Variant`: represented as object, using the `{constructor: body}` format, e.g. `{"Left": true}`.

8.1.2.13 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_transacion_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.

8.1.2.14 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`:

```
daml 1.2 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`:

```
daml 1.2 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 contract instances of first version of the template which were created before the new field is added, and there are contract instances 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 contract instances:

owner	provider	accountid	displayname	observers
Bob	Bob	1239-4321	Personal	["Alice"]
Bob	Bob	6021-5678	NULL	["Alice"]

8.1.2.15 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

8.1.2.16 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).

8.1.2.17 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.

8.1.2.18 Troubleshooting

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$@[kka.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.

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.

8.2 DAML Integration Kit - ALPHA

8.2.1 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*.

Please note that currently the tool is in experimental status.

8.2.1.1 Downloading the tool

Run the following command to fetch the tool:

```
curl -L 'https://bintray.com/api/v1/content/digitalassetsdk/
↪DigitalAssetSDK/com/daml/ledger/testtool/ledger-api-test-tool_2.12/
↪$latest/ledger-api-test-tool_2.12-$latest.jar?bt_package=sdk-components'
↪-o ledger-api-test-tool.jar
```

This will create a file `ledger-api-test-tool.jar` in your current directory.

8.2.1.2 Extracting `.dar` files required to run the tests

Before you can run the Ledger API test tool on your ledger, you need to load a specific set of DAML templates onto your ledger.

1. To obtain the corresponding `.dar` files, run:

   ```
   $ java -jar ledger-api-test-tool.jar --extract
   ```

   This writes all `.dar` files required for the tests into the current directory.

2. Load all `.dar` files into your Ledger.

8.2.1.3 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
```

If any test embedded in the tool fails, it will print out details of the failure for further debugging.

8.2.1.4 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 arguments runs the *default tests*. Use the following command line flags to select which tests to run:

> `--list`: print all available tests to the console
> `--include`: only run the tests provided as argument
> `--exclude`: do not run the tests provided as argument
> `--all-tests`: run all default and optional tests. This flag can be combined with the `--exclude` flag.

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 default 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 --all-tests localhost:6865
```

Listing 4: Run all tests, but not `TestC`

```
$ java -jar ledger-api-test-tool.jar --all-tests --exclude TestC
```

8.2.1.5 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
```

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.

8.2.1.6 Testing your tool from continuous integration pipelines

To test your ledger in a CI pipeline, run it as part of your pipeline:

```
$ java -jar ledger-api-test-tool localhost:6865 2>&1 /dev/null
$ echo $?
0
```

The tool is tailored to be used in CI pipelines: as customary, when the tests succeed, it will produce minimal output and return the success exit code.

8.2.1.7 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.

We used this flag during tool development to ensure that the tool does not always return success.

8.2.1.8 Tuning the testing behaviour of the tool

Use the command line options `--timeout-scale-factor` and `--command-submission-ttl-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.
> Set `--command-submission-ttl-scale-factor` to adjust the time-to-live of commands as represented by the MRT (Maximum Record Time) on the Ledger API. The default value is 1.0 and will be applied to the default TTL, which is the maximum TTL as returned by the LedgerConfigurationService. In any case, the used TTL value will be clipped to stay between the minimum and maximum TTL.

DAML Applications run on DAML Ledgers. A DAML Ledger is a server serving the *Ledger API* as per the semantics defined in the *DA 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.

8.2.2 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

8.2.3 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*.

8.2.3.1 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 *Quickstart guide*.
2. Get an in-depth understanding of the *DA 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*.

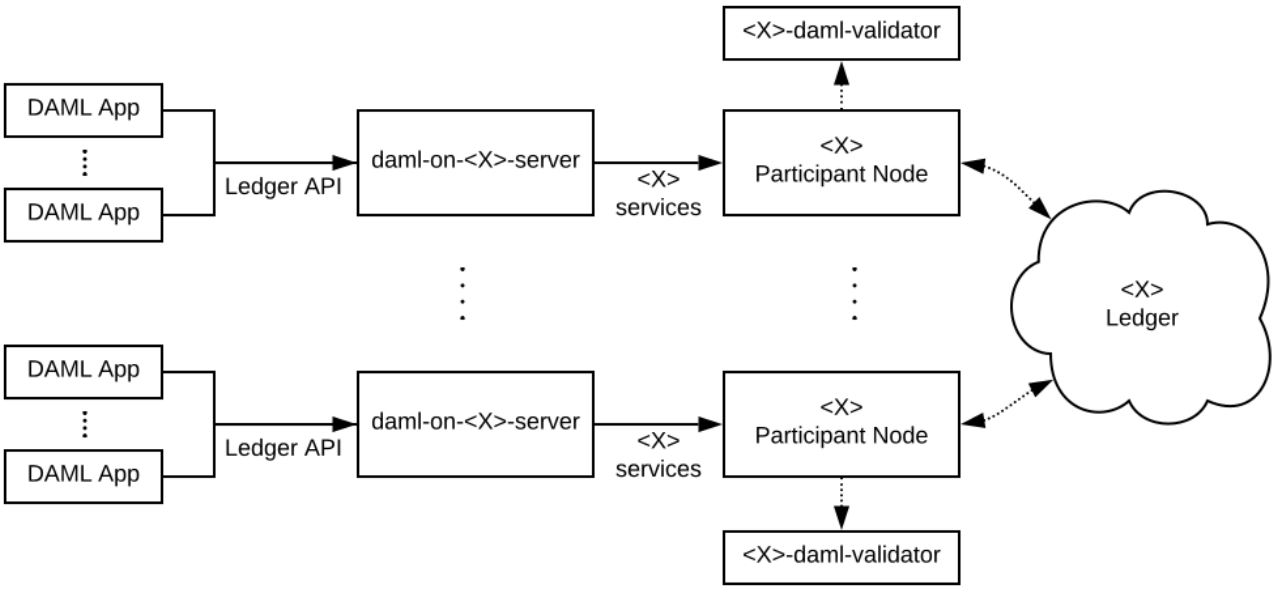
If you need help, then feel free to use the feedback form on this documentation page or GitHub issues on the digital-asset/daml repository to get into contact with us.

8.2.3.2 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)

8.2.3.3 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 the DAML SDK. 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.

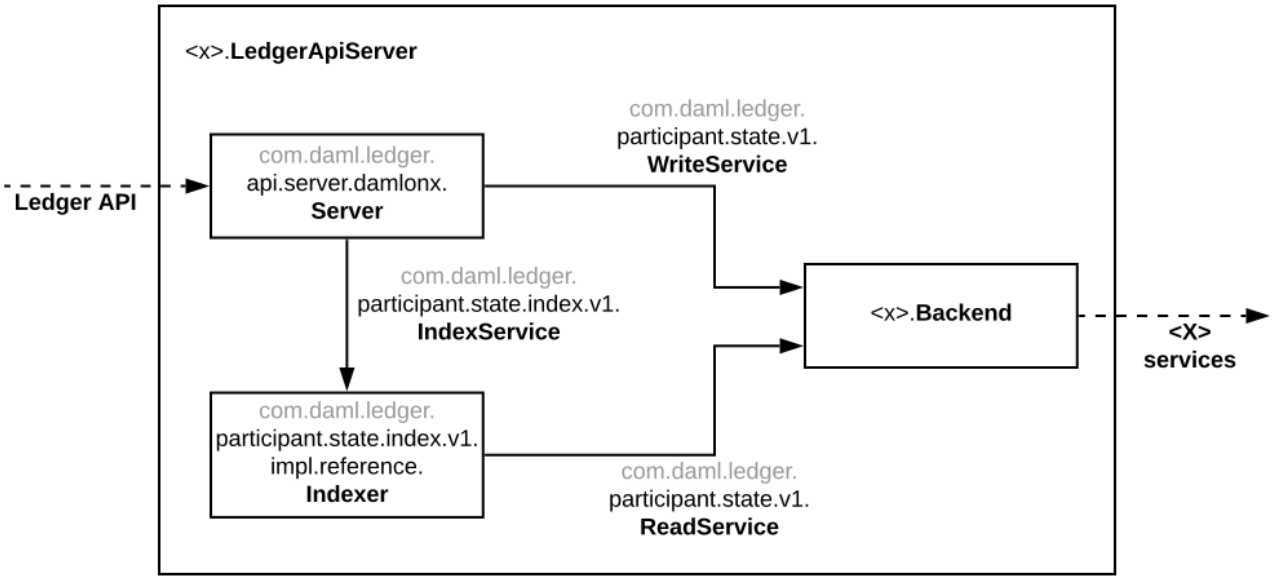
`participant-state-kvutils.jar` (source code) These utilities provide methods to succintly implement interfaces from `participant-state.jar` on top of a key-value state storage.
See documentation in package.scala

`ledger-api-server.jar` (source code for 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`.

8.2.4 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).

8.2.5 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. Obtain the tool:
```
curl -L 'https://bintray.com/api/v1/content/digitalassetsdk/
DigitalAssetSDK/com/daml/ledger/testtool/ledger-api-test-tool_2.
12/$latest/ledger-api-test-tool_2.12-$latest.jar?bt_package=sdk-
components' -o ledger-api-test-tool.jar
```
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*.

8.2.6 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).

8.3 HTTP JSON API Service

WARNING: the HTTP JSON API described in this document is actively being designed and is *subject to breaking changes*, including all request and response elements demonstrated below or otherwise implemented by the API. We welcome feedback about the API on our issue tracker or on Slack.

The JSON API provides a significantly simpler way than *the Ledger API* to access *basic active contract set functionality*:

> creating contracts,
> exercising choices on contracts, and
> querying the current active contract set.

The goal is to get you up and running writing effective ledger-integrated applications 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
> ledger metaprogramming (e.g. packages and templates).

For these and other features, use *the Ledger API* instead.

8.3.1 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.

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.

8.3.1.1 ContractId

Contract ids are expressed as their string representation:

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

8.3.1.2 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.

8.3.1.3 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`.

8.3.1.4 Timestamp

Input

Timestamps are represented as ISO 8601 strings, rendered using the format `yyyy-mm-ddThh:mm:ss[.ssssss]Z`:

```
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
```

It's OK to omit the microsecond part partially or entirely. 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.

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.

8.3.1.5 Party

Represented using their string representation, without any additional quotes:

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

8.3.1.6 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.

8.3.1.7 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, 9999-99-99].

8.3.1.8 Text

Represented as strings.

8.3.1.9 Bool

Represented as booleans.

8.3.1.10 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.

8.3.1.11 List

Lists are represented as

```
[v1, ..., v]
```

8.3.1.12 Map

Maps are represented as objects:

```
{ k1: v1, ..., k: v }
```

8.3.1.13 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, always applying the shortcut for None record fields.

8.3.1.14 Variant

Variants are expressed as

```
{ constructor: 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:

```
{"Bar": 42}
{"Baz": {}}
{"Quux": null}
{"Quux": 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

```
{"Bar": {"f1": 42, "f2": true}}
{"Baz": {}}
```

This can be encoded and used in TypeScript, including exhaustiveness checking; see a keyed example.

8.3.1.15 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.

8.3.2 /contracts/search query language

The body of `POST /contracts/search` looks like so:

```
{"%templates": [...template IDs...],
 ...other query elements...}
```

The elements of that query are defined here.

8.3.2.1 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.

8.3.2.2 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" }`

Example: `{ favorites: ["vanilla", "chocolate"] }`

> Match: `{ favorites: ["vanilla", "chocolate"] }`
> No match: `{ favorites: ["chocolate", "vanilla"] }`
> No match: `{ favorites: ["vanilla", "strawberry"] }`
> No match: `{ favorites: ["vanilla", "chocolate", "strawberry"] }`

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 the forthcoming comparison queries.

8.3.2.3 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:

```
{"%templates": [{"moduleName": "Foo", "entityName": "A"},
                {"moduleName": "Foo", "entityName": "B"},
                {"moduleName": "Foo", "entityName": "C"}],
 "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.

8.3.3 How to start

8.3.3.1 Start sandbox from a DAML project directory

```
$ daml sandbox --wall-clock-time --ledgerid MyLedger ./.daml/dist/
↪quickstart-0.0.1.dar
```

8.3.3.2 Start HTTP service from a DAML project directory

```
$ daml json-api --ledger-host localhost --ledger-port 6865 \
    --http-port 7575 --max-inbound-message-size 4194304 --package-reload-
↪interval 5s \
    --application-id HTTP-JSON-API-Gateway
```

```
$ daml json-api --help
HTTP JSON API daemon
Usage: http-json-binary [options]
```

(continues on next page)

(continued from previous page)

```
  --help                    Print this usage text
  --ledger-host <value>     Ledger host name or IP address
  --ledger-port <value>     Ledger port number
  --address <value>         IP address that HTTP JSON API service listens
↪on. Defaults to 0.0.0.0.
  --http-port <value>       HTTP JSON API service port number
  --application-id <value>
                            Optional application ID to use for ledger
↪registration. Defaults to HTTP-JSON-API-Gateway
  --package-reload-interval <value>
                            Optional interval to poll for package updates.
↪Examples: 500ms, 5s, 10min, 1h, 1d. Defaults to 5 seconds
  --max-inbound-message-size <value>
                            Optional max inbound message size in bytes.
↪Defaults to 4194304
```

8.3.4 Example session

```
$ daml new iou-quickstart-java quickstart-java
$ cd iou-quickstart-java/
$ daml build
$ daml sandbox --wall-clock-time --ledgerid MyLedger ./.daml/dist/
↪quickstart-0.0.1.dar
$ daml json-api --ledger-host localhost --ledger-port 6865 --http-port 7575
```

8.3.4.1 Choosing a party

You specify your party and other settings with JWT. In testing environments, you can use https://jwt.io to generate your token.

The default header is fine. Under Payload, fill in:

```
{
  "ledgerId": "MyLedger",
  "applicationId": "foobar",
  "party": "Alice"
}
```

Keep in mind: - the value of `ledgerId` payload field has to match `--ledgerid` passed to the sandbox. - you can replace `Alice` with whatever party you want to use.

Under Verify Signature, put `secret` as the secret (_not_ base64 encoded); that is the hardcoded secret for testing.

Then the Encoded box should have your token; set HTTP header `Authorization: Bearer copy-paste-token-here`.

Here are two tokens you can use for testing:

```
{"ledgerId": "MyLedger", "applicationId": "foobar",
"party": "Alice"}                    eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
```

```
eyJsZWRnZXJJZCI6Ik15TGVkZ2VyIiwiYXBwbGljYXRpb25JZCI6ImZvb2JhciIsInBhcnR5IjoiQWx
4HYfzjlYr1ApUDot0a6a4zB49zS_jrwRUOCkAiPMqo0
{"ledgerId": "MyLedger", "applicationId": "foobar",
"party": "Bob"}                    eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJsZWRnZXJJZCI6Ik15TGVkZ2VyIiwiYXBwbGljYXRpb25JZCI6ImZvb2JhciIsInBhcnR5IjoiQm9
2LE3fAvUzLx495JWpuSzHye9YaH3Ddt4d2Pj0L1jSjA
```

For production use, we have a tool in development for generating proper RSA-encrypted tokens locally, which will arrive when the service also supports such tokens.

8.3.4.2 GET http://localhost:7575/contracts/search

8.3.4.3 POST http://localhost:7575/contracts/search

List currently active contracts that match a given query.

application/json body, formatted according to the */contracts/search query language*:

```
{"%templates": [{"moduleName": "Iou", "entityName": "Iou"}],
 "amount": 999.99}
```

empty output:

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

output, each contract formatted according to *DAML-LF JSON Encoding*:

```
{
    "status": 200,
    "result": [
        {
            "observers": [],
            "agreementText": "",
            "signatories": [
                "Alice"
            ],
            "contractId": "#489:0",
            "templateId": {
                "packageId":
↪"ac3a64908d9f6b4453329b3d7d8ddea44c83f4f5469de5f7ae19158c69bf8473",
                "moduleName": "Iou",
                "entityName": "Iou"
            },
            "witnessParties": [
                "Alice"
            ],
            "argument": {
                "observers": [],
                "issuer": "Alice",
                "amount": "999.99",
```

(continues on next page)

(continued from previous page)

```
                "currency": "USD",
                "owner": "Alice"
            }
        }
    ]
}
```

8.3.4.4 POST http://localhost:7575/command/create

Create a contract.

application/json body, `argument` formatted according to *DAML-LF JSON Encoding*:

```
{
  "templateId": {
    "moduleName": "Iou",
    "entityName": "Iou"
 },
  "argument": {
    "observers": [],
    "issuer": "Alice",
    "amount": "999.99",
    "currency": "USD",
    "owner": "Alice"
  }
}
```

output:

```
{
    "status": 200,
    "result": {
        "observers": [],
        "agreementText": "",
        "signatories": [
            "Alice"
        ],
        "contractId": "#56:0",
        "templateId": {
            "packageId":
↪"ac3a64908d9f6b4453329b3d7d8ddea44c83f4f5469de5f7ae19158c69bf8473",
            "moduleName": "Iou",
            "entityName": "Iou"
        },
        "witnessParties": [
            "Alice"
        ],
        "argument": {
            "observers": [],
            "issuer": "Alice",
```

(continues on next page)

(continued from previous page)

```
            "amount": "999.99",
            "currency": "USD",
            "owner": "Alice"
        }
    }
}
```

8.3.4.5 POST http://localhost:44279/command/exercise

Exercise a choice on a contract.

`"contractId": "#56:0"` is the value from the create output application/json body:

```
{
    "templateId": {
        "moduleName": "Iou",
        "entityName": "Iou"
    },
    "contractId": "#56:0",
    "choice": "Iou_Transfer",
    "argument": {
        "newOwner": "Alice"
    }
}
```

output:

```
{
    "status": 200,
    "result": [
        {
            "archived": {
                "contractId": "#56:0",
                "templateId": {
                    "packageId":
↪"ac3a64908d9f6b4453329b3d7d8ddea44c83f4f5469de5f7ae19158c69bf8473",
                    "moduleName": "Iou",
                    "entityName": "Iou"
                },
                "witnessParties": [
                    "Alice"
                ]
            }
        },
        {
            "created": {
                "observers": [],
                "agreementText": "",
                "signatories": [
                    "Alice"
```

(continues on next page)

(continued from previous page)

```
                ],
                "contractId": "#301:1",
                "templateId": {
                    "packageId":
↪"ac3a64908d9f6b4453329b3d7d8ddea44c83f4f5469de5f7ae19158c69bf8473",
                    "moduleName": "Iou",
                    "entityName": "IouTransfer"
                },
                "witnessParties": [
                    "Alice"
                ],
                "argument": {
                    "iou": {
                        "observers": [],
                        "issuer": "Alice",
                        "amount": "999.99",
                        "currency": "USD",
                        "owner": "Alice"
                    },
                    "newOwner": "Alice"
                }
            }
        }
    ]
}
```

8.3.4.6 GET http://localhost:7575/parties

output:

```
{
    "status": 200,
    "result": [
        {
            "party": "Alice",
            "isLocal": true
        }
    ]
}
```

8.4 DAML Triggers - Off-Ledger Automation in DAML

8.4.1 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.

8.4.1.1 Module Daml.Trigger

Data Types

data *ACS*

Active contract set, you can use `getTemplates` to access the templates of a given type.

instance HasField acs (TriggerState s) *ACS*

instance HasField initialize (*Trigger* s) (*ACS* -> s)

instance HasField rule (*Trigger* s) (Party -> *ACS* -> Map *CommandId* [*Command*] -> s -> *TriggerA* ())

instance HasField updateState (*Trigger* s) (*ACS* -> *Message* -> s -> s)

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	*ACS* -> s	Initialize the user-defined state based on the ACS.
updateState	*ACS* -> *Message* -> s -> s	Update the user-defined state based on the ACS and a transaction or completion message.
rule	Party -> *ACS* -> Map *CommandId* [*Command*] -> s -> *TriggerA* ()	The rule defines the main logic of your trigger. Given the party your trigger is running as, the ACS, the commands in flight and the user-defined state, you can send commands to the ledger using `emitCommands` to change the ACS.

instance HasField initialize (*Trigger* s) (*ACS* -> s)

instance HasField rule (*Trigger* s) (Party -> *ACS* -> Map *CommandId* [*Command*] -> s -> *TriggerA* ())

instance HasField updateState (*Trigger* s) (*ACS* -> *Message* -> s -> s)

data *TriggerA* 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 Functor *TriggerA*

instance Action *TriggerA*

instance Applicative *TriggerA*

instance HasField rule (*Trigger* s) (Party -> *ACS* -> Map *CommandId* [*Command*] -> s -> *TriggerA* ())

Functions

getTemplates : Template a => *ACS* -> [(ContractId a, a)]
Extract the templates of a given type from the ACS.

emitCommands : [*Command*] -> *TriggerA CommandId*
Send a transaction consisting of the given commands to the ledger.

dedupCreate : (*Eq* t, Template t) => t -> *TriggerA* ()
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.

dedupExercise : (*Eq* c, Choice t c r) => ContractId t -> c -> *TriggerA* ()
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`.

8.4.1.2 Module Daml.Trigger.LowLevel

Data Types

data *ActiveContracts*

ActiveContracts

Field	Type	Description
activeContracts	[*Created*]	

instance HasField activeContracts *ActiveContracts* [*Created*]

instance HasField initialState (*Trigger* s) (Party -> *ActiveContracts* -> (s, [*Commands*]))

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 *Show AnyContractId*

instance HasField contractId *AnyContractId* (ContractId ())

instance HasField contractId *Archived AnyContractId*

instance HasField contractId *Command AnyContractId*

instance HasField contractId *Created 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	

instance HasField choiceArg *Command* AnyChoice

instance HasField commands *Commands* [*Command*]

instance HasField commandsInFlight TriggerAState (Map *CommandId* [*Command*])

instance HasField commandsInFlight (TriggerState s) (Map *CommandId* [*Command*])

instance HasField contractId *Command AnyContractId*

instance HasField rule (*Trigger* s) (Party -> *ACS* -> Map *CommandId* [*Command*] -> s -> *TriggerA* ())

instance HasField templateArg *Command* AnyTemplate

data *CommandId*

CommandId Text

instance Eq *CommandId*

instance Show *CommandId*

instance HasField commandId *Commands CommandId*

instance HasField commandId *Completion CommandId*

instance HasField commandId *Transaction* (Optional *CommandId*)

instance HasField commandsInFlight TriggerAState (Map *CommandId* [*Command*])

instance HasField commandsInFlight (TriggerState s) (Map *CommandId* [*Command*])

instance HasField rule (*Trigger* s) (Party -> *ACS* -> Map *CommandId* [*Command*] -> s -> *TriggerA* ())

instance MapKey *CommandId*

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*]

instance HasField emittedCommands TriggerAState [*Commands*]

instance HasField initialState (*Trigger* s) (Party -> *ActiveContracts* -> (s, [*Commands*]))

instance HasField update (*Trigger* s) (*Message* -> s -> (s, [*Commands*]))

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.

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.

MTransaction Transaction

MCompletion Completion

instance HasField update (*Trigger* s) (*Message* -> s -> (s, [*Commands*]))

instance HasField updateState (*Trigger* s) (*ACS* -> *Message* -> s -> s)

data *TemplateId*

We implicitly assume that the package id corresponds to the package the trigger is part of. This type is temporary until we have a builtin in LF for identifiers.

TemplateId

Field	Type	Description
moduleName	Text	
entityName	Text	

instance HasField entityName *TemplateId* Text

instance HasField moduleName *TemplateId* Text

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* -> (s, [*Commands*])	
update	*Message* -> s -> (s, [*Commands*])	

instance HasField initialState (*Trigger* s) (Party -> *ActiveContracts* -> (s, [*Commands*]))

instance HasField update (*Trigger* s) (*Message* -> s -> (s, [*Commands*]))

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.

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.

fromCreate : Template t => *Command* -> Optional t
Check if the command corresponds to a create 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.

WARNING: DAML Triggers are an experimental feature that is actively being designed and is *subject to breaking changes*. We welcome feedback about DAML triggers on our issue tracker or on Slack.

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, e.g., it creates a template of another type.

It is possible to write these clients in a language of your choice, e.g., 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.

8.4.2 Usage

Our example for this tutorial consists of 3 templates.

First, we have a template called `Original`:

```
template Original
  with
    owner : Party
    name : Text
    textdata : Text
  where
    signatory owner

    key (owner, name) : (Party, Text)
    maintainer key._1
```

This template has an `owner`, a `name` that identifies it and some `textdata` that we just represent as `Text` to keep things simple. We have also added a contract key to ensure that each owner can only have one `Original` with a given `name`.

Second, we have a template called `Subscriber`:

```
template Subscriber
  with
    subscriber : Party
    subscribedTo : Party
  where
    signatory subscriber
    observer subscribedTo
    key (subscriber, subscribedTo) : (Party, Party)
    maintainer key._1
```

This template allows the `subscriber` to subscribe to `Original` s where `subscribedTo` is the `owner`. For each of these `Original` s, our DAML trigger should then automatically create an instance of third template called `Copy`:

```
template Copy
  with
    original : Original
```

(continues on next page)

(continued from previous page)

```
    subscriber : Party
  where
    signatory (signatory original)
    observer subscriber
```

Our trigger should also ensure that the `Copy` contracts stay in sync with changes on the ledger. That means that we need to archive `Copy` contracts if there is more than one for the same `Original`, we need to archive `Copy` contracts if the corresponding `Original` has been archived and we need to archive all `Copy` s for a given subscriber if the corresponding `Subscriber` contract has been archived.

8.4.2.1 Implementing a DAML Trigger

Having defined what our DAML trigger is supposed to do, we can now move on to its implementation. 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-triggers` library to the `dependencies` field in `daml.yaml`.

```
dependencies:
  - daml-prim
  - daml-stdlib
  - daml-trigger
```

In addition to that you also need to import the `Daml.Trigger` module.

DAML triggers automatically track the active contract set 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 copy 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 : ACS -> s
  , updateState : ACS -> Message -> s -> s
  , rule : Party -> ACS -> Map CommandId [Command] -> s -> TriggerA ()
  }
```

The `initialize` function is called on startup and allows you to initialize your user-defined state based on 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.

Finally, 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, the commands in flight and the user defined state. The type `TriggerA` allows you to emit commands that are then sent to the ledger. Like `Scenario` or `Update`, you can use `do` notation with `TriggerA`.

For our DAML trigger, the definition looks as follows:

```
copyTrigger : Trigger ()
copyTrigger = Trigger
  { initialize = \_acs -> ()
  , updateState = \_acs _message () -> ()
  , rule = copyRule
  }
```

Now we can move on to the most complex part of our DAML trigger, the implementation of `copyRule`. First let's take a look at the signature:

```
copyRule : Party -> ACS -> Map CommandId [Command] -> () -> TriggerA ()
copyRule party acs commandsInFlight () = do
```

We will need the party and the ACS to get the `Original` contracts where we are the owner, the `Subscriber` contracts where we are in the `subscribedTo` field and the `Copy` contracts where we are the `owner` of the corresponding `Original`.

The commands in flight will be useful to avoid sending the same command multiple times if `copyRule` is run multiple times before we get the corresponding transaction. Note that DAML triggers are expected to be designed such that they can cope with this, e.g., after a restart or a crash where the commands in flight do not contain commands in flight from before the restart, so this is an optimization rather than something required for them to function correctly.

First, we get all `Subscriber`, `Original` and `Copy` contracts from the ACS. For that, the DAML trigger API provides a `getTemplates` function that given the ACS will return a list of all contracts of a given type.

```
  let subscribers : [(ContractId Subscriber, Subscriber)] = getTemplates
↪@Subscriber acs
  let originals : [(ContractId Original, Original)] = getTemplates
↪@Original acs
  let copies : [(ContractId Copy, Copy)] = getTemplates @Copy acs
```

Now, we can filter those contracts to the ones where we are the `owner` as described before.

```
  let ownedSubscribers = filter (\(_, s) -> s.subscribedTo == party)
↪subscribers
  let ownedOriginals = filter (\(_, o) -> o.owner == party) originals
  let ownedCopies = filter (\(_, c) -> c.original.owner == party) copies
```

We also need a list of all parties that have subscribed to us.

```
  let subscribingParties = map (\(_, s) -> s.subscriber) ownedSubscribers
```

As we have mentioned before, we only want to keep one `Copy` per `Original` and `Subscriber` and archive all others. Therefore, we group identical `Copy` contracts and keep the first of each group while archiving the others.

```
  let groupedCopies : [[(ContractId Copy, Copy)]]
      groupedCopies = groupOn snd $ sortOn snd $ ownedCopies
  let copiesToKeep = map head groupedCopies
  let archiveDuplicateCopies = concatMap tail groupedCopies
```

In addition to duplicate copies, we also need to archive copies where the corresponding `Original` or `Subscriber` no longer exists.

```
  let archiveMissingOriginal = filter (\(_, c) -> c.original `notElem` map
↪snd ownedOriginals) copiesToKeep
  let archiveMissingSubscriber = filter (\(_, c) -> c.subscriber `notElem`
↪subscribingParties) copiesToKeep
  let archiveCopies = dedup $ map fst $ archiveDuplicateCopies <>
↪archiveMissingOriginal <> archiveMissingSubscriber
```

To send the corresponding archve commands to the ledger, we iterate over `archiveCopies` using `forA` and call the `emitCommands` function. Each call to `emitCommands` takes a list of commands which will be submitted as a single transaction. The actual commands can be created using `exerciseCmd` and `createCmd`.

```
  forA archiveCopies $ \cid -> emitCommands [exerciseCmd @Copy cid Archive]
```

Finally, we also need to create copies that do not already exists. We want to avoid creating copies for which there is already a command in flight. The DAML Trigger API provides a `dedupCreate` helper for this which only sends the commands if it is not already in flight.

```
  let neededCopies = [Copy m o | (_, m) <- ownedOriginals, o <-
↪subscribingParties]
  let createCopies = filter (\c -> c `notElem` map snd copiesToKeep)
↪neededCopies
  mapA dedupCreate createCopies
```

8.4.2.2 Running a DAML Trigger

To try this example out, you can replicate it using `daml new copy-trigger copy-trigger`. You first have to build the trigger like you would build a regular DAML project using `daml build`. Then start the sandbox and navigator using `daml start`.

Now we are ready to run the trigger using `daml trigger`:

```
daml trigger --dar .daml/dist/copy-trigger-0.0.1.dar --trigger-name
↪CopyTrigger:copyTrigger --ledger-host localhost --ledger-port 6865 --
↪ledger-party Alice
```

The first argument specifies the `.dar` file that we have just built. The second argument specifies the identifier of the trigger using the syntax `ModuleName:identifier`. Finally, we need to specify the ledger host, port and the party that our trigger is executed as.

Now open Navigator at http://localhost:7500/.

First, login as `Alice` and create an `Original` contract with `party` set to `Alice`. Now, logout and login as `Bob` and create a `Subscriber` contract with `subscriber` set to `Bob` and `subscribedTo` set to `Alice`. After a short delay you should now see a `Copy` contract corresponding to the `Original` that you have created as `Alice`. Once you archive the `Subscriber` contract, you can see that the `Copy` contract will also be archived.

8.4.3 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. If your automation needs to interact with data outside of the ledger then DAML triggers are not the right tool. For this case, you can use the HTTP JSON API.

Chapter 9

Support and updates

9.1 Support

Have questions or feedback? You're in the right place.

> **Questions: Stack Overflow**
> For how do I?, why does something work this way or I've got a programming problem I'm trying to solve questions, the daml tag on Stack Overflow is the best place to ask.
> If you're not sure what makes a good question, take a look at this checklist.
> **Help and feedback: Slack**
> If you want to give feedback, or have questions that aren't right for Stack Overflow, you can join the DAML community on Slack and talk to us in the `#public` channel.

When you're in the community Slack or on Stack Overflow, please keep to the Code of Conduct.

9.1.1 Support expectations

For community users (ie on Slack 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 only offer public support - for example, on the `#public` channel in Slack. We can't answer questions in private messages or over email, so please only ask questions in the `#public` channel.
> **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 DA about how to build your DAML application, we offer paid support. Please contact us to ask about pricing.

9.2 Release notes

This page contains release notes for the SDK.

9.2.1 0.13.34 - 2019-11-07

9.2.1.1 DAML-LF - Internal

Freeze DAML-LF 1.7. Summary of changes (See DAML-LF specification for more details.):

- Add support for parametrically scaled Numeric type.
- Drop support of Decimal in favor or Numerics.
- Add interning of strings and names. This reduces drastically dar file size.
- Add support for 'Any' type.
- Add support for type representation values.

Add immutable bintray/maven packages for handling DAML-LF archive up to version 1.7:

- *com.digitalasset.daml-lf-1.7-archive-proto*
  This package contains the archive protobuf definitions as they were introduced when 1.7 was frozen. These definitions can be used to read DAML-LF archives up to version 1.7.

9.2.1.2 DAML Triggers

Triggers must now be compiled with `daml build --target 1.7` instead of `1.dev`.

9.2.2 0.13.33 - 2019-11-06

9.2.2.1 Navigator

Fixed regression in Navigator to properly respect the CLI option `--ledger-api-inbound-message-size-max` again. See issue #3301.

9.2.2.2 DAML Compiler

Reduce the memory footprint of the IDE and the command line tools (ca. 18% in our experiments).
Fix compile error caused by instantiating generic templates at `Numeric n`.
The compiler now accepts single-constructor enum types. For example `data A = A` or `data Foo = Bar`.

9.2.2.3 DAML Triggers

Add `dedupCreate` and `dedupExercise` helpers that will only send commands if they are not already in flight.
Remove the custom `AbsoluteContractId` type in favor of the regular `ContractId` type used in DAML templates.

9.2.2.4 Sandbox

Fixed a bug a database migration script for Sandbox on Postgres introduced in SDK 0.13.32. See issue #3284.
Timing about database operations are now exposed over JMX as well as via the logs.
Added a missing index to the SQL schema for the Postgres Ledger.

9.2.2.5 DAML Integration Kit

Re-add *integration kit documentation* that got accidentally deleted.

9.2.2.6 Ledger API

Disallow empty commands. See issue #592.

9.2.2.7 DAML Stdlib

Add *DA.TextMap.filter* and *DA.Next.Map.filter*.
Add *assertEq* and *assertNotEq* to *DA.Assert* as synonyms for *===* and *=/=*.
Add `DA.Foldable.mapA_`, `DA.Foldable.forA_`, `DA.Foldable.sequence_` and `DA.Action.replicateA_`. These functions match the behavior of corresponding functions without the underscore suffix but ignore the result which can be more convenient and efficient.

9.2.2.8 Extractor - Experimental

Extractor now stores exercise events in the single table data format. See issue #3274.

9.2.2.9 JSON API - Experimental

`workflowId` no longer included in any responses.
`/contracts/search` endpoint can optionally store searched contracts in a Postgres-based cache, by passing the new `--query-store-jdbc-config` option. See issue #2781.

9.2.2.10 DAML SDK

Display release notes in the IDE when the DAML extension is upgraded.

9.2.3 0.13.32 - 2019-10-29

9.2.3.1 DAML Triggers

The trigger runner now supports triggers using the high-level API directly. These no longer need to be converted to low-level Triggers using `runTrigger`. Triggers using the low-level API are still supported.
The trigger runner has a new command that just lists the triggers in a dar using `daml trigger list --dar path/to/dar`.

9.2.3.2 DAML Compiler

The package database is now be cleaned automatically on initialization. This means that you should no longer have to run `daml clean` on SDK upgrades if you use DAR dependencies (e.g. with DAML triggers).

9.2.3.3 Sandbox

Improve performance of looking up contracts from postgres. See issue #2330.

9.2.4 0.13.31 - 2019-10-18

9.2.4.1 Sandbox

Party management fix, see issue #3177.

The maximum allowed TTL for commands is now configurable via the `--max-ttl-seconds` parameter, for example: `daml sandbox --max-ttl-seconds 300`.
Fixed a bug where `CreatedEvent#event_id` field is not properly filled by `ActiveContractsService`. See issue #65.

9.2.4.2 DAML SDK

Shrink docker image containing the full DAML SDK from 2.8 GB to 1.2 GB.

9.2.4.3 Navigator

Accept and use an access token to be used against Ledger API servers that require authentication, see issue #3156.
Demo-oriented password workflow has been removed.

9.2.4.4 Ledger Client

Expose new method to construct channels for more granular control over the client creation process.

9.2.4.5 JSON API - Experimental

Add `/parties` endpoint.

9.2.4.6 DAML Triggers - Experimental

The trigger runner now logs output from `trace`, `error` and failed command completions and hides internal debugging output.

9.2.4.7 DAML-LF - Internal

Changed the name of the bintray/maven package from `com.digitalasset.daml-lf-archive-scala` to `com.digitalasset.daml-lf-archive-reader`

9.2.5 0.13.30 - 2019-10-15

9.2.5.1 DAML Standard Library

Add `DA.Action.State` module containing a `State` action that can be used for computations that modify a state variable.
Add `createAndExercise`.

9.2.5.2 DAML Compiler

Fixed the location of interface files when the `source` field in `daml.yaml` points to a file. This is mainly important for when you want to use the created `.dar` in the `dependencies` field of another package. See issue #3135.

9.2.5.3 DAML-LF

Breaking Rename DAML-LF Archive protobuf package from *com.digitalasset.daml_lf* to *com.digitalasset.daml_lf_dev*. This will only affect you do not use the DAML-LF Archive reader provided with the SDK but a custom one based on code generation by protoc.
Breaking Some bintray/maven packages are renamed:

- *com.digitalasset.daml-lf-proto* becomes *com.digitalasset.daml-lf-dev-archive-proto*

- *com.digitalasset.daml-lf-archive* becomes *com.digitalasset:daml-lf-dev-archive-java-proto*‘

Add immutable bintray/maven packages for handling DAML-LF archive up to version 1.6 in a stable v

- *com.digitalasset.daml-lf-1.6-archive-proto*
  This package contains the archive protobuf definitions as they were introduced when 1.6 was frozen. These definitions can be used to read DAML-LF archives up to version 1.6.
  The main advantage of this package over the *dev* version (*com.digitalasset.daml-lf-dev-archive-proto*) is that it is immutable (it is guaranteed to never changed once introduced in the SDK). In other words one can used it without suffering frequent breaking changes introduced in the *dev* version.
  Going forward the SKD will contain a similar immutable package containning the proto definition for at least each DAML-LF version the compiler supports.
  We strongly advise anyone reading DAML-LF Archive directly to use this package (or the *com.digitalasset:daml-lf-1.6-archive-java-proto* package described below). Breaking changes to the *dev* version may be introduced frequently and without further notice in the release notes.
- *com.digitalasset:daml-lf-1.6-archive-java-proto*
  This package contains the java classes generated from the package *com.digitalasset.daml-lf-1.6-archive-proto*

9.2.5.4 DAML Triggers

This release contains a first version of an experimental DAML triggers feature that allows you to implement off-ledger automation in DAML.

9.2.5.5 DAML-SDK Docker Image

The image now contains a `daml` user and the SDK is installed to `/home/daml/.daml`. `/home/daml/.daml/bin` is automatically added to `PATH`.

9.2.5.6 JSON API - Experimental

Support for automatic package reload See issue #2906.

9.2.5.7 Java Bindings

Add helper to prepare transformer for `Bot.wire`. See issue #3097.

9.2.5.8 Ledger

The ledger api index server starts only after the indexer has finished initializing the database.

9.2.5.9 Sandbox

Filter contracts or contracts keys in the database query for parties that cannot see them.

9.2.5.10 Scala Bindings

Fixed a bug in the retry logic of `LedgerClientBinding#retryingConfirmedCommands`. Commands are now only retried when the server responds with status `RESOURCE_EXHAUSTED` or `UNAVAILABLE`.

9.2.5.11 Scala Codegen

Fixes for StackOverflowErrors in reading large LF archives. See issue #3104.

9.2.5.12 SQL Extractor

The format used for storing Optional and Map values found in contracts as JSON has been replaced with *DAML-LF JSON Encoding*. See issue #3066 for specifics.

9.2.6 0.13.29 - 2019-10-04

Rerelease of 0.13.28 since that failed due to CI issues.

9.2.7 0.13.28 - 2019-10-04

9.2.7.1 JSON API - Experimental

Returning archived and active/created contracts from `/command/exercise` endpoint. See issue #2925.
Flattening the output of the `/contracts/search` endpoint. The endpoint returns `ActiveContract` objects without `GetActiveContractsResponse` wrappers. See issue #2987.

9.2.7.2 SDK

Bundle the `daml-trigger` package. Note, this package is experimental and will change.
Releases can now bundle additional libraries with the SDK in `$DAML_SDK/daml-libs`. You can refer to them in your `daml.yaml` file by listing the package name without `.dar` extension. See issue #2979.

9.2.7.3 DAML Studio

`damlc ide` now also supports a `--target` option. The easiest way to specify this is the `build-options` field in `daml.yaml`.
Fix a bug where the same module was imported twice under different file paths caused module name collisions. See issue #3099.

9.2.7.4 Ledger

Improve SQL backend performance by eliminating extra queries to the database.
Enhance logging to correlate log messages with the associated participant id in multi-participant node tests and environments
Ledger api server indexer closes akka system on shutdown.
The ledger api server now stores divulged, otherwise unknown contracts.

9.2.7.5 DAML Visualization

Adding *daml damlc visual-web* command. visual-command generates webpage with d3 network.

9.2.7.6 DAML Ledger Integration Kit

The transaction service is now fully tested.
The TTL for commands is now read from the configuration service.
The contract key tests now live under a single test suite and are multi-node aware.

9.2.7.7 DAML Compiler

Fix a problem where constraints of the form `Template (Foo t)` caused the compiler to suggest enabling the `UndecidableInstances` language extension.
Generic template instantiations like `template instance IouProposal = Proposal Iou` now generate a type synonym `type IouProposal = Proposal Iou` that can be used in DAML. Before, they generated a `newtype`, which cannot be used anymore.
Fixed a bug where `damlc build` sometimes did not find modules during typechecking even if they were present during parallel compilations.

9.2.7.8 Security

Document how to verify the signature on release tarballs.

9.2.8 0.13.27 - 2019-09-25

9.2.8.1 DAML Assistant

`daml start` now supports `--sandbox-option=opt`, `--navigator-option=opt` and `--json-api-option=opt` to pass additional option to sandbox/navigator/json-api. These flags can be specified multiple times.

9.2.8.2 DAML Compiler

Fix a bug where generic templates could crash the compiler.

9.2.8.3 Security

Fix signing process.

9.2.9 0.13.26 - 2019-09-24

9.2.9.1 JSON API

`/contracts/search` now supports a query language for filtering the contracts returned by matching fields. See issue 2778.

9.2.9.2 DAML Compiler

Fix a bug where `.dar` files produced by `daml build` were missing all `.daml` files except for the one that `source` pointed to.
Fix a bug where importing the same module from different directories resulted in an error in `daml build`.
`damlc migrate` now produces a project that can be built with `daml build` as opposed to having to use the special `build.sh` and `build.cmd` scripts.

9.2.9.3 DAML Integration Toolkit

30 more test cases have been added to the transaction service test suite.

9.2.9.4 Security

Starting with this one, releases are now signed on GitHub.

9.2.10 0.13.25 - 2019-09-18

9.2.10.1 Documentation

Suppress instance documentation when *–data-only* mode is requested.

9.2.10.2 DAML-LF

Add CAST_NUMERIC and SHIFT_NUMERIC in DAML-LF 1.dev.
Change signature of MUL_NUMERIC and DIV_NUMERIC.

9.2.10.3 DAML Integration Kit

Fix contract key uniqueness check in kvutils.
Preload packages in a background thread in kvutils.

9.2.10.4 Ledger

ActiveContractsService now specifies to always return at least one message with the offset. This removes a special case where clients would need to check if the stream was empty or not.
Dramatically increased performance of the ActiveContractService by only loading the contracts that the parties in the transaction filter are allowed to see.

9.2.11 0.13.24 - 2019-09-16

9.2.11.1 Java codegen

If the DAR source cannot be read, the application crashes and prints an error report.

9.2.11.2 DAML Assistant

Java and Scala codegen is now integrated with the assistant and distributed with the SDK. It can be run via `daml codegen`. You can find more information in the DAML Assistant documentation.

9.2.11.3 DAML Compiler

Fix bug with qualified imports of generic templates.

9.2.11.4 Ledger

Upgraded ledger-api server H2 Database version to 1.4.199 with stability fixes including one to the `merge` statement.

9.2.11.5 DAML Integration Kit

One more test case added. Transaction service tests are not multi-node aware.
Semantic tests now ensure synchronization across participants when running in a multi-node setup.

9.2.12 0.13.23 - 2019-09-11

9.2.12.1 DAML Integration Kit

The reference implementation can now spin up multiple nodes, either scaling a single participant horizontally or adding new participants. Check the CLI `--help` option.

The test tool now runs the double spend test on a shared contract in a multi-node setup (as well as single-node).
The test tool can now run all semantic test in a multi-node setup.

9.2.12.2 DAML Standard Library

BREAKING CHANGE The `(/)` operator was moved out of the `Fractional` typeclass into a separate `Divisible` typeclass, which is now the parent class of `Fractional`. The `Int` instance of `Fractional` is discontinued, but there is an `Int` instance of `Divisible`. This change will break projects that rely on the `Fractional Int` instance. To fix that, change the code to rely on `Divisible Int` instead. This change will also break projects where a `Fractional` instance is defined. To fix that, add a `Divisible` instance and move the definition of `(/)` there.

9.2.12.3 DAML Assistant

The HTTP JSON API is now integrated with the assistant and distributed with the SDK. It can either be launched via `daml json-api` or via `daml start`. You can find more information in the README.
The *daml.yaml* file now supports an additional field *build-options*, which you can use to list cli options you want added to invocations of *daml build* and *daml ide*.

9.2.12.4 JSON API

BREAKING CHANGE The `/contracts/search` request payload must use `"%templates"` in place of `"templateIds"` to select which templates' contracts are returned. See issue #2777.

9.2.12.5 DAML Compiler

BREAKING CHANGE Move the DAML-LF produced by generic template instantiations closer to the surface syntax. See the documentation on How DAML types are translated to DAML-LF for details.

9.2.13 0.13.22 - 2019-09-04

9.2.13.1 DAML Assistant

BREAKING CHANGE Changed the meaning of the `source` field in the daml.yaml file to be a pointer to the source directory of the DAML code contained in a project relative to the project root. This is breaking projects, where the `source` field of the project is pointing to a non-toplevel location in the source code directory structure.

9.2.13.2 DAML Integration Kit

Introduced initial support for multi-node testing. Note that for the time being no test actually uses more than one node.
BREAKING CHANGE The `-p` / `--target-port` and `-h` / `--host` flags have been discontinued. Pass one (or more) endpoints to test as command line arguments in the `<host>:<port>` form.

9.2.13.3 Documentation

Basic explanation of generic templates.

9.2.13.4 Ledger API

BREAKING CHANGE In Protobuf `Value` message, rename `decimal` field to `numeric`.

9.2.13.5 Sandbox

Updated the PostgreSQL JDBC driver to version 42.2.6.
Added TRACE level debugging for database operations.
Fixed a bug that could lead to an inconsistent snapshot of active contracts being served by the ActiveContractsService under high load.
Commands are now deduplicated based on `(submitter, application_id, command_id)`.

9.2.14 0.13.21 - 2019-08-29

9.2.14.1 DAML Compiler

Enable the language extension `FlexibleContexts` by default.
BREAKING CHANGE Enable the language extension `MonoLocalBinds` by default. `let` and `where` bindings introducing polymorphic functions that are used at different types now need an explicit type annotation. Without the type annotation the type of the first use site will be inferred and use sites at different types will fail with a type mismatch error.

9.2.14.2 Java Codegen

Fix bug that caused the generation of duplicate methods that affected sources with data constructors with type parameters that are either non-unique or not presented in the same order as in the corresponding data type declaration. See #2367.

9.2.14.3 Ledger

H2 Database support in the Ledger API Server.

9.2.14.4 Sandbox

The sandbox now properly sets the connection pool properties `minimumIdle`, `maximumPoolSize`, and `connectionTimeout`.

9.2.15 0.13.20 - 2019-08-22

9.2.15.1 Documentation

Added platform-independent tips for testing

9.2.15.2 DAML Compiler

Some issues that caused `damlc test` to crash on shutdown have been fixed.
The DAML compiler was accidentally compiled without optimizations on Windows. This has been fixed which should improve the performance of `damlc` and `daml studio` on Windows.
`damlc build` should no longer leak file handles so `ulimit` workarounds should no longer be necessary.
Allow more contexts in generic templates. Specifically, template constraints can have arguments besides type variables, if the FlexibleContexts extension is enabled.

9.2.15.3 DAML-LF

Breaking Rename `NUMERIC` back to `DECIMAL` in Protobuf definition.

9.2.15.4 DAML Studio

`damlc ide` now also accepts `--ghc-option` arguments like `damlc build` so `damlc ide --ghc-option -W` launches the IDE with more warnings.
The VSCode extension now has a configuration field for passing extra arguments to `damlc ide`.

9.2.15.5 DAML Integration Kit

Participant State API and kvutils was extended with support for changing the ledger configuration. See changelog in respective `package.scala` files.

9.2.15.6 Sandbox

Fixed a bug that caused the reset service to hang for 10 seconds. See issue #2549.

9.2.15.7 Java Bindings

The Java Codegen now supports parametrized ContractIds. See #2258.

9.2.15.8 DAML Standard Library

Add `stripInfix` function to `DA.List`.

9.2.16 0.13.19 - 2019-08-14

9.2.16.1 Sandbox

Fixed a bug that prevented the ledger from loading transactions with empty workflow ids.
Fixed internal shutdown order to avoid dead letter warnings when stopping Sandbox/Ledger API Server. See issue #1886.

9.2.16.2 DAML Studio

Added a new command for visualizing a project in the IDE.
Print stack trace when a scenario fails.
Various memory leaks have been fixed so long-running sessions should no longer show a significant increase in memory usage.

9.2.16.3 DAML Compiler

The `--project-root` option now works properly with relative paths in `daml build`.
Support generic template declarations and instances. Documentation for generic templates is still being worked on.
The `--dump-pom` flag from `damlc package` has been removed as packaging has not relied on POM files for a while.

9.2.16.4 Navigator

`{"None": {}}` and `{"Some": value}`, where previously accepted, are no longer supported or used for DAML `Optional` values. Instead, for simple cases, use the plain value for `Some`, and `null` for `None`. See issue #2361 for other cases.

9.2.16.5 HTTP JSON API

A new, more intuitive JSON format for DAML values is supported. See issue #2361.

9.2.17 0.13.18 - 2019-08-07

Fix a bug where `daml studio` did not launch VSCode on Windows.

9.2.18 0.13.17 - 2019-08-07

9.2.18.1 DAML Docs

For `damlc docs`, the `--template` argument now takes the path to a Mustache template when generating Markdown, Rst, and HTML output. The template can use `title` and `body` variables to control the appearance of the docs.

9.2.18.2 DAML Assistant

Spaces in user names or other parts of file names should now be handled correctly.
The `daml deploy` and `daml ledger` experimental commands were added. Use `daml deploy --help` and `daml ledger --help` to find out more about them.

9.2.19 0.13.16 - 2019-08-01

9.2.19.1 DAML Compiler

BREAKING CHANGE Handwritten instances of `Template` and `Choice` typeclasses are no longer supported. All template constructs must be defined using declarations inside `template` syntax.

9.2.19.2 DAML Docs

The `damlc docs` command now produces docs to a folder by default. Use the new `--combine` flag to output a single file instead.
The `damlc docs` flag `--prefix` has been replaced with a `--template` flag which allows for a more flexible template.
The `damlc docs` flag `--json` has been dropped in favor of `--format=json`.

9.2.19.3 Extractor

BREAKING CHANGE Changed schema to accomodate removed field `ExercisedEvent#contract_creating_event_id`. Existing database schemas are not compatible anymore with the newer version. The extractor needs to be run on an empty schema from Ledger Begin.

9.2.19.4 Java Bindings

Add all packages of java bindings to the javadocs. See #2280.
BREAKING CHANGE Removed field `ExercisedEvent#contract_creating_event_id`. See #2068.

9.2.19.5 Ledger API

BREAKING CHANGE Removed field `ExercisedEvent#contract_creating_event_id`. See #2068.

9.2.19.6 Sandbox

The active contract service correctly serves stakeholders. See #2070.

Added the `--maxInboundMessageSize` CLI parameter to set the maximux size of messages received through the Ledger API. If the value is not set the current default is preserved (4 MB). Makes package uploads idempotent and tolerate partial duplicates. See #2130.

9.2.20 0.13.15 - 2019-07-25

9.2.20.1 DAML Studio

Scenario links no longer disappear if the current file does not compile. The location is adjusted but this is done one a best effort basis and can fail if the scenario itself is modified.

9.2.20.2 DAML Compiler

Support reading of DAML-LF 1.5 again.

9.2.20.3 DAML-LF

Breaking Rename `DECIMAL` by `NUMERIC` in archive Protobuf definition.

9.2.20.4 Ledger API

BREAKING: Drop support for legacy identifier. The previously deprecated field `name` in `Identifier` message is not supported anymore. Use `module_name` and `entity_name` instead.

9.2.20.5 Navigator

Fixed an issue when Navigator console did not see any contracts. See #2271.

9.2.20.6 Documentation

Improved the Maven pom.xml file for `quickstart-java` to better integrate with VS Code. See #887.

9.2.20.7 Releases

Releases should now be announced on the releases blog.

9.2.21 0.13.14 - 2019-07-22

9.2.21.1 DAML Compiler

Support reading of DAML-LF 1.5 again.

9.2.21.2 DAML Studio

VSCode scenario view improvements. Add a note in the IDE if:

there is an open scenario view for a scenario that does no longer exist,
there is an open scenario view for a scenario in a file that does no longer compile.

9.2.22 0.13.13 - 2019-07-16

9.2.22.1 DAML Assistant

Fix VSCode path for use if not already in PATH on mac
BREAKING: remove *-replace=newer* option.

9.2.22.2 DAML Studio

Fix a bug where the extension seemed to disappear every other time VS Code was opened.
DAML Studio now displays a Processing indicator on the bottom left while the IDE is doing work in the background.

9.2.22.3 Sandbox

Fixing an issue around handling passTime in scenario loader See #1953.
Remembering already loaded packages after reset See #1979.

9.2.22.4 DAML-LF

Release version 1.6. This versions provides:

- `enum` types. See issue #105 and DAML-LF 1 specification for more details.
- new builtins for (un)packing strings. See issue #16.
- intern package IDs. See issue #1614.

9.2.22.5 DAML Compiler

Add support for DAML-LF `1.6`. In particular:

- **BREAKING CHANGE** Add support for `enum` types. DAML variant types that look like enumerations (i.e., those variants without type parameters and without arguments) are compiled to the new DAML-LF `enum` type when DAML-LF 1.6 target is selected. For instance the daml type declaration of the form:

  ```
  data Color = Red | Green | Blue
  ```

  will produce a DAML-LF `enum` type instead of DAML-LF `variant` type. This change is breaking, since this release makes DAML-LF `1.6` the default compiler output.
- Add `DA.Text.toCodePoints` and `DA.Text.fromCodePoints` primitives to (un)pack strings.
- Add support for DAML-LF intern package IDs.

BREAKING CHANGE Make DAML-LF 1.6 the default output. This change activates the support of `enum` type describes above.
BREAKING CHANGE Drop support for DAML-LF 1.5. Compiling to DAML-LF 1.6 requires some changes regarding enum types to applications using the Ledger API, see above. (The ledger server still supports DAML-LF 1.5.)

9.2.22.6 Ledger API

Add support for `enum` types. Simple DAML `variant` types will be mapped to DAML-LF `enum` types when using a DAML-LF `1.6` archive. Ledger API Value Protobuf provides the new `Enum` message. This message must be used to communicate this new data type throught the API.

9.2.22.7 Java Codegen

Add support for `enum` types. `enum` types are mapped to standard java enum. See Generate Java code from DAML for more details.

9.2.22.8 Scala Codegen

Add support for `enum` types.

9.2.22.9 Navigator

Add support for `enum` types.

9.2.22.10 Extractor

Add support for `enum` types.

9.2.22.11 DAML Docs

Added links to type signatures in generated docs. Check out the updated standard library docs.

9.2.23 0.13.12 - 2019-07-09

9.2.23.1 DAML Assistant

Fix VSCode path for use if not already in PATH on mac.
Kill child processes on `SIGTERM`. This means that killing `daml sandbox` will also kill the sandbox process.

9.2.23.2 DAML-LF

Fixed regression that produced an invalid daml-lf-archive artefact. See #2058.

9.2.23.3 DAML Docs

BREAKING CHANGE `damlc docs` now typechecks the source files before doc generation, to be able to use type information during doc generation. This may break existing doc builds.
Added `--package-name` and `--input-format` flags to `damlc docs`.

9.2.24 0.13.11 - 2019-07-08

9.2.24.1 Sandbox

The completion stream method of the command completion service uses the ledger end as a default value for the offset. See #1913.
Fixed an issue when CompletionService returns offsets having inclusive semantics when used for re-subscription. See #1932.
DAML-LF packages used by the sandbox are now stored in Postgres, allowing users to resume a Postgres sandbox ledger without having to again specify all packages through the CLI. See #1929.

9.2.24.2 Java Bindings

Added overloads to the Java bindings `CompletionStreamRequest` constructor and the `CommandCompletionClient` to accept a request without an explicit ledger offset. See #1913.
DEPRECATION: the `CompletionStreamRequest#getOffset` method is deprecated in favor of the non-nullable `CompletionStreamRequest#getLedgerOffset`. See #1913.

9.2.24.3 Scala Bindings

Contract keys are exposed on CreatedEvent. See #1681.

9.2.24.4 Navigator

Contract keys are show in the contract details page. See #1681.

9.2.24.5 DAML Standard Library

BREAKING CHANGE: Remove the deprecated modules `DA.Map`, `DA.Set`, `DA.Experimental.Map` and `DA.Experimental.Set`. Please use `DA.Next.Map` and `DA.Next.Set` instead.
Add `Sum` and `Product` newtypes that provide `Monoid` instances based on the `Additive` and `Multiplicative` instances of the underlying type.
Add `Min` and `Max` newtypes that provide `Semigroup` instances based `min` and `max`.

9.2.24.6 DAML Compiler

The default output path for all artifacts is now in the `.daml` directory. In particular, the default output path for .dar files in `daml build` is now `.daml/dist/<projectname>.dar`.

9.2.24.7 DAML Studio

DAML Studio is now published as an extension in the Visual Studio Code marketplace. The `daml studio` command will now install the published extension by default, but will revert to the extension bundled with the DAML SDK if installation fails. You can get the old default behavior of always using the bundled extension by running `daml studio --replace=newer` or `daml studio --replace=always` instead.
You can now configure the gRPC message size limit in `daml.yaml` via `scenario-service: {"grpc-max-message-size": 1000000}`. This will set the limit to 1000000 bytes. This should only be necessary for very large projects.
You can now configure the gRPC timeout `daml.yaml` via `scenario-service: {"grpc-timeout": 42}`. This option will set the timeout to 42 seconds. You should only need to set this option for very large projects.

9.2.24.8 DAML Integration Kit

Make DivulgenceIT properly work when run via the Ledger API Test Tool.
The submission service shuts down its ExecutorService upon exit to ensure a smooth shutdown.

9.2.24.9 DAML-LF

The DAML-LF development version (`1.dev`) includes a new, breaking restriction regarding contract key lookups. In short, when looking up or fetching a key, the transaction submitter must be one of the key maintainers. Note that this change is not breaking since the compiler does not produce DAML-LF `1.dev` by default. However it will be a breaking change once this restriction makes it into DAML-LF `1.6` and once DAML-LF `1.6` becomes the default.

9.2.25 0.13.10 - 2019-06-28

9.2.25.1 Sandbox

Added *-log-level* command line flag.
BREAKING CHANGE: The Sandbox no longer supports loading from DALF files. You can now only use DAR files. See #1610.

9.2.25.2 Ledger API

Added new CLI flags `--stable-party-identifiers` and `--stable-command-identifiers` to the *Ledger API Test Tool* to allow disabling randomization of party and command identifiers. It is useful for testing of ledgers which are configured with a predefined static set of parties.

9.2.26 0.13.9 - 2019-06-28

9.2.26.1 DAML Studio

Fix an error in the `package.json` that stopped the extension from being loaded.

9.2.27 0.13.8 - 2019-06-27

9.2.27.1 Navigator

Contract details now show signatories and observers. See #1269.

9.2.27.2 Scala Bindings

Reflect addition of signatories and observers to the bindings. See #1269.

9.2.27.3 Java Codegen

Generated code supports signatories and observers as exposed by the bindings. See #1269.

9.2.27.4 Java Bindings

Reflect addition of signatories and observers to the bindings. See #1269.

9.2.27.5 Ledger API

Expose signatories and observers for a contract in `CreatedEvent`. See #1269.
BREAKING CHANGE: Specify pretty C# namespaces in ledger api protos. C# bindings will end up in a different namespace than the default one. See #1901.

9.2.27.6 DAML Compiler

BREAKING CHANGE: Drop support for DAML-LF 1.4. Compiling to DAML-LF 1.5 should work without any code changes, although we highly recommend not specifying a target DAML-LF version at all. (The ledger server still supports DAML-LF 1.4.)

9.2.27.7 Sandbox

Made the archive CLI arguments optional. See #1905.

9.2.27.8 DAML-LF

BREAKING CHANGE: Specify pretty C# namespaces in archive protos. C# bindings will end up in a different namespace than the default one. See #1900.

9.2.28 0.13.7 - 2019-06-26

9.2.28.1 DAML-LF

Rename `none` and `some` to `optional_none` and `optional_some`, resp., in `Expr` and `CasePat`.

9.2.29 0.13.6 - 2019-06-25

9.2.29.1 DAML Assistant

Added `--install-assistant` flag to `daml install` command, changing the default behavior of `daml install` to install the assistant whenever we are installing a newer version of the SDK. Deprecated the `--activate` flag.
Added `--start-navigator`, `--on-start`, and `--wait-for-signal` options to `daml start`, to make scripting and testing with the sandbox much easier.

9.2.29.2 DAML Studio

Opening an already open scenario will now focus it rather than opening it in a new empty tab which is never updated with results.
The selected view for scenario results (table or transaction) is now preserved when the scenario results are updated. See #1675.
Goto definition now works on the export list of modules.
Goto definition now works on types.

9.2.29.3 DAML-LF

Rename `TO_TEXT_CODE_POINTS` and `FROM_TEXT_CODE_POINTS` to `TEXT_FROM_CODE_POINTS` and `TEXT_TO_CODE_POINTS`, resp.

9.2.29.4 Dependencies

Protobuf has been upgraded to version 3.8.0. This also includes the protobuf-java library used as a dependency.

9.2.29.5 Ledger API

Added additional Ledger API integration tests to Ledger API Test Tool.

9.2.29.6 Java Bindings

The artefact `com.daml.ledger:bindings-java` now has `grpc-netty` as dependency so that users don't need to explicitly add it.

9.2.29.7 DAML Integration Kit

Fixed a bug in the test tool that prevented users from running the tests. See #1841

9.2.29.8 Navigator

Added support for SDK project configuration files. If you start Navigator with the SDK Assistant, Navigator will directly read the `daml.yaml` config file instead of the old Navigator config file. See #1128.

9.2.29.9 Docker Image

The daml-sdk docker images are now based on Alpine Linux.

9.2.30 0.13.5 - 2019-06-19

9.2.30.1 Release Procedure

Fixes to the CI/CD release procedure. See *#1755 <https://github.com/digital-asset/daml/issues/1755>__*.

9.2.30.2 Sandbox

Introduced a new API for package management. See #1311.

9.2.31 0.13.4 - 2019-06-19

9.2.31.1 Java Codegen

Support generic types (including tuples) as contract keys in codegen. See #1728.

9.2.31.2 Ledger API

A new command `ExerciseByKey` allows to exercise choices on active contracts referring to them by their key. See #1366.

9.2.31.3 Java Bindings

The addition of the `ExerciseByKey` to the Ledger API is reflected in the bindings. See #1366.

9.2.31.4 Release Procedure

Fixes to the release procedure. Note: The release to Maven Central was successfully performed _manually_ in release 0.13.3. This release should confirm that it will occur as part of the CI/CD. See #1745

9.2.31.5 DAML Studio

Closing and reopening scenario results will now show the results instead of an empty view. See #1606.

9.2.32 0.13.3 - 2019-06-18

9.2.32.1 Release Procedure

Fixes to the release procedure. See #1737

9.2.32.2 Java Bindings

The changes for Java Bindings listed for SDK 0.13.2 now only apply to SDK 0.13.3 and later. This is due to the partial failure of the release procedure.

9.2.32.3 Docs

Added *An introduction to DAML*

9.2.32.4 DAML Studio

The IDE now executes tasks in parallel.

9.2.32.5 Sandbox

Fixed a bug in migration scripts that could cause databases originally created with older versions of the Sandbox to not upgrade schemas properly. See #1682.

9.2.33 0.13.2 - 2019-06-18

9.2.33.1 Visualizing DAML Contracts

Added Visualizing DAML Contracts

9.2.33.2 Release Procedure

Fixes to the release procedure. See #1725
The changes for Java Bindings listed for SDK 0.13.1 now only apply to SDK 0.13.2 and later. This is due to the partial failure of the release procedure.

9.2.34 0.13.1 - 2019-06-17

9.2.34.1 Language

Add an instance for `IsParties (Optional Party)`, allowing `Optional` values to be used in `signatory`, `observer` and `maintainer` clauses.

9.2.34.2 Java Bindings

Release the Java Bindings to the public Maven Central repository. To move to using the Maven Central repository, remove the `<repository>...</repository>` and `<pluginRepository>...</pluginRepository>` blocks from Maven POM files that use version 0.13.1 (or later) of the Java Bindings. See #1205.

9.2.35 0.13.0 - 2019-06-17

9.2.35.1 SDK

This marks the first release that is no longer released for the `da` assistant. It is still possible to use it to get older SDK releases. Take a look at documentation for the new `daml` assistant for migration instructions.

9.2.35.2 Sandbox

Fixed a bug in an internal data structure that broke contract keys. See #1623.
Fixed a bug of not closing a resource properly when shutting down the Sandbox. See #1702.

9.2.35.3 DAML Studio

Double the gRPC message limit used for the scenario service. This avoids issues on large projects.

9.2.35.4 Ledger API

Slash (/) is now an allowed character in contract, workflow, application and command identifiers.

9.2.36 0.12.25 — 2019-06-13

9.2.36.1 DAML Integration Kit

Added new CLI flag `--all-tests` to the *Ledger API Test Tool* to run all default and optional tests.
Added new CLI flag `--command-submission-ttl-scale-factor` to the *Ledger API Test Tool*. It scales time-to-live of commands sent for ledger processing (captured as Maximum Record Time in submitted transactions) for some suites. Useful to tune Maximum Record Time depending on the environment and the Ledger implementation under test.
Fixed various bugs in the daml-on-x ledger api server and index service.

9.2.36.2 Sandbox

Introduced a new API for party management. See #1312.

9.2.36.3 Scala bindings

New *-root* command-line option for limiting what templates are selected for codegen. See #1210.

9.2.36.4 Ledger API

Contract keys are now available for created events from the transaction service. See #1268.

9.2.36.5 Java Bindings

The addition of contract keys on created events in the Ledger API is reflected in the bindings. See #1268.

9.2.36.6 Java Codegen

Contracts decoded from the transaction service now expose their contract key (if defined). See #1268.

9.2.37 0.12.24 - 2019-06-06

9.2.37.1 DAML Studio

Fix errors due to unhandled `$/cancelRequest` and `textDocument/willSave` requests from showing up in the output tab in VSCode. These errors also caused an automatic switch from the problems tab to the output tab which should now no longer happen.
Note that upgrading the VSCode extension requires launching it via `daml studio`. If you launch VSCode directly, you might get issues due to an outdated extension.

9.2.38 0.12.23 - 2019-06-05

9.2.38.1 SQL Extractor

50MiB is no longer hard-coded on extractor input for sandbox or any other server, permitting large packages; e.g. pass `--ledger-api-inbound-message-size-max 62914560` to extractor to get a 60MiB limit. See #1520.
Improving logging. See #1518.

9.2.38.2 DAML Language

BREAKING CHANGE: Contract key maintainers must now explicitly be computed from the contract key using the implicit `key` variable. For instance, if you have `key (bank, accountId) : (Party, Text)` and want `bank` to be the maintainer, you have to write `maintainer key._1` (before, you could write `maintainer bank`).

9.2.38.3 DAML Compiler

BREAKING CHANGE: Drop support for DAML-LF 1.3. Compiling to DAML-LF 1.4 should work without any code changes, although we highly recommend not specifying a target DAML-LF version at all. (The ledger server still supports DAML-LF 1.3.)
Fix initialization of package-db for non-default DAML-LF versions. This fixes issues when using daml build –target 1.3 (or other target versions).

9.2.38.4 DAML Standard Library

Add `enumerate` function.

9.2.38.5 Navigator

Fixed a regression where Navigator console was not able to inspect contracts and events. See #1454.
50MiB is no longer hard-coded on extractor input for sandbox or any other server, permitting large packages; e.g. pass `--ledger-api-inbound-message-size-max 62914560` to extractor to get a 60MiB limit. See #1520.

9.2.38.6 Sandbox

Added recovery around failing ledger entry persistence queries using Postgres. See #1505.

9.2.38.7 DAML Integration Kit

The *Ledger API Test Tool* can now optionally run `TransactionServiceIT` as part of the conformance tests. This means you need to load additional `.dar` files into the ledger under test. Please refer to the updated instructions in the *documentation*.
Added new CLI options to the *Ledger API Test Tool*:

- `--list` prints all available tests to the console
- `--include` takes a comma-separated list of test names that should be run
- `--exclude` takes a comma-separated list of test names that should not be run

9.2.39 0.12.22 - 2019-05-29

9.2.39.1 DAML Studio

Fixed a bug where type check errors would persist if there was a subsequent parse error.

9.2.39.2 DAML Compiler

BREAKING CHANGE: Drop support for DAML-LF 1.2. Compiling to DAML-LF 1.3 should work without any code changes, although we highly recommend not specifying a target DAML-LF version at all.
BREAKING CHANGE: By default `damlc test` must be executed in a project and will test the whole project. Testing individual files, potentially outside a project, requires passing the new `--files` flag.

9.2.39.3 DAML-LF

The Syntax of party literals is relaxed by allowing the character colon. Concretely those literals must match the regular expression `[a-zA-Z0-9:\-_ ]+` instead of `[a-zA-Z0-9\-_ ]+` previously. See #1467.

9.2.39.4 SQL Extractor

The extractor `--party` option may now specify multiple parties, separated by commas; e.g. instead of `--party Bob` you can say `--party Bob,Bar,Baz` and get the contracts for all three parties in the database. See #1360.
The extractor `--templates` option to specify template IDs in the format: `<module1>:<entity1>,<module2>:<entity2>`. If not provided, extractor subscribes to all available templates. See #1352.

9.2.39.5 Sandbox

Fixed a bug in the SQL backend that caused transactions with a fetch node referencing a contract created in the same transaction to be rejected. See issue #1435.

9.2.40 0.12.21 - 2019-05-28

9.2.40.1 DAML Assistant

The `exposed-modules` field in `daml.yaml` is now optional. If it is not specified, all modules in the project are exposed. See #1328.
You can now see all available versions with `daml version` using the `--all` flag.

9.2.40.2 DAML Compiler

BREAKING CHANGE: Drop support for DAML-LF 1.1. Compiling to DAML-LF 1.2 should work without any code changes, although we highly recommend not specifying a target DAML-LF version at all.
Make DAML-LF 1.5 the default version produced by the compiler.

9.2.40.3 DAML Standard Library

`parseInt` and `parseDecimal` now work at more extremes of values and accept leading plus signs.

9.2.40.4 DAML-LF

Add new version 1.5. See DAML-LF 1 specification for details.

9.2.40.5 Ledger

BREAKING CHANGE: The string fields `application_id`, `command_id`, `ledger_id`, and `workflow_id` in Ledger API commands must now match the regular expression `[A-Za-z0-9\._:\-#]{1,255}`. Those fields were unrestricted UTF-8 strings in previous versions. See #398.

9.2.41 0.12.20 - 2019-05-23

9.2.41.1 Sandbox

Contract keys: Support arbitrary key expressions (this was accidentally omitted from 0.12.19).

9.2.42 0.12.19 - 2019-05-22

9.2.42.1 Ledger

Transaction filters in *GetTransactionsRequest* without any party are now rejected with *INVALID_ARGUMENT* instead of yielding an empty stream
See #1250 for details.

9.2.42.2 DAML

Contract keys: The syntactic restriction on contract keys has been removed. They can be arbitray expressions now.

9.2.42.3 DAML-LF

Add new version 1.4 and make it the default version produced by `damlc`. It removes the syntactic restriction on contract keys.

9.2.42.4 Java Bindings

Bots: A class called LedgerTestView was added to make bot unit testing possible

9.2.42.5 DAML

BREAKING CHANGE - Syntax: Records with empty update blocks, e.g. `foo with`, is now an error (the fact it was ever accepted was a bug).
BREAKING CHANGE - Contract Keys: Before, maintainers were incorrectly not checked to be a subset of the signatories, now they are. See issue #1123

9.2.42.6 Sandbox

When loading a scenario with `--scenario`, the sandbox no longer compiles packages twice, see issue #1238.
When starting the sandbox, you can now choose to have it load all the `.dar` packages immediately with the `--eager-package-loading` flag. The default behavior is to load the packages only when a command requires them, which causes a delay for the first command that requires a yet-to-be-compiled package. See issue #1230.

9.2.42.7 SDK tools

The Windows installer is now signed. You might still see Windows defender warnings for some time but the publisher should now show Digital Asset Holdings, LLC.

9.2.43 0.12.18 - 2019-05-20

9.2.43.1 Documentation

Removed unnecessary dependency in the quickstart-java example project.
Removed the *Configure Maven* section from the installation instructions. This step is not needed anymore.

9.2.43.2 SDK tools

DAML Assistant: We've built a new and improved version of the SDK assistant, replacing `da` commands with `daml` commands. The documentation is updated to use the new assistant in this release.

For a full guide to what's changed and how to migrate, see *Moving to the new DAML assistant*. To read about how to use the new `daml` Assistant, see *DAML Assistant (daml)*.

9.2.43.3 DAML

BREAKING CHANGE - DAML Compiler: It is now an error to omit method bodies in class `instance` s if the method has no default. Almost all instances of such behaviour were an error - add in a suitable definition.
Contract keys: We've added documentation for contract keys, a way of specifying a primary key for contract instances. For information about how to use them, see *Contract keys*.
BREAKING CHANGE - DAML Standard Library: Moved the `Tuple` and `Either` types to `daml-prim:DA.Types` rather than exposing internal locations.
How to migrate:
- You don't need to change DAML code as a result of this change.
- People using the Java/Scala codegen need to replace `import ghc.tuple.*` or `import da.internal.prelude.*` with `import da.types.*`.
- People using the Ledger API directly need to replace `GHC.Tuple` and `DA.Internal.Prelude` with `DA.Types`.

BREAKING CHANGE - DAML Standard Library: Don't expose the `TextMap` type via the `Prelude` anymore.
How to migrate: Always import `DA.TextMap` when you want to use the `TextMap` type.
DAML Standard Library: Add `String` as a compatibility alias for `Text`.

9.2.43.4 Ledger API

BREAKING Removed the unused field *ExercisedEvent* from *Event*, because a *Transaction* never contains exercised events (only created and archived events): #960
This change is *backwards compatible on the transport level*, meaning:
- new versions of ledger language bindings will work with previous versions of the Sandbox, because the field was never populated
- previous versions of the ledger language bindings will work with new versions of the Sandbox, as the field was removed without any change in observable behavior

How to migrate:
- If you check for the presence of *ExercisedEvent* when handling a *Transaction*, you have to remove this code now.

Added the *agreement text* as a new field `agreement_text` to the `CreatedEvent` message. This means you now have access to the agreement text of contracts via the Ledger API. The type of this field is `google.protobuf.StringValue` to properly reflect the optionality on the wire for full backwards compatibility. See Google's wrappers.proto for more information about `StringValue`.
See #1110 for details.
Fixed: the *CommandService.SubmitAndWait* endpoint no longer rejects commands without a workflow identifier.
See #572 for details.

9.2.43.5 Java Bindings

BREAKING Reflect the breaking change of Ledger API in the event class hierarchy:
- Changed `data.Event` from an abstract class to an interface, representing events in a flat transaction.
- Added interface `data.TreeEvent`, representing events in a transaction tree.
- `data.CreatedEvent` and `data.ArchivedEvent` now implement `data.Event`.

- `data.CreatedEvent` and `data.ExercisedEvent` now implement `data.TreeEvent`.
- `data.TransactionTree#eventsById` is now `Map<String, TreeEvent>` (was previously `Map<String, Event>`).

How to migrate:

- If you are processing `data.TransactionTree` objects, you need to change the type of the processed events from `data.Event` to `data.TreeEvent`.
- If you are checking for the presense of exercised events when processing `data.Transaction` objects, you can remove that code now. It would never have triggered in the first place, as transactions do not contain exercised events.

Java Codegen: You can now call a method to get a `CreateAndExerciseCommand` for each choice, for example:

```
CreateAndExerciseCommand cmd = new MyTemplate(owner, someText).
↪createAndExerciseAccept(42L);
```

In this case `MyTemplate` is a DAML template with a choice `Accept` and the resulting command will create a contract and exercise the `Accept` choice within the same transaction.
See issue #1092 for details.
Added *agreement text* of contracts: #1110

- **Java Bindings**
  * Added field `Optional<String> agreementText` to `data.CreatedEvent`, to reflect the change in Ledger API.
- **Java Codegen**
  * Added generated field `Optional<String> TemplateName.Contract#agreementText`.
  * Added generated static method `TemplateName.Contract.fromCreatedEvent(CreatedEvent)`. This is the preferred method to use for converting a `CreatedEvent` into a `Contract`.
  * Added generated static method `TemplateName.Contract.fromIdAndRecord(String, Record, Optional<String>)`. This method is useful for setting up tests, when you want to convert a `Record` into a contract without having to create a `CreatedEvent` first.
  * Deprecated generated static method `TemplateName.Contract.fromIdAndRecord(String, Record)` in favor of the new static methods in the generated `Contract` classes.
  * Changed the generated *decoder utility class* to use the new `fromCreatedEvent` method.
  * **BREAKING** Changed the return type of the `getDecoder` method in the generated decoder utility class from `Optional<BiFunction<String, Record, Contract>>` to `Optional<Function<CreatedEvent, Contract>>`.

How to migrate:

- If you are manually constructing instances of `data.CreatedEvent` (for example, for testing), you need to add an `Optional<String>` value as constructor parameter for the `agreementText` field.
- You should change all calls to `Contract.fromIdAndRecord` to `Contract.fromCreatedEvent`.

```
// BEFORE
CreatedEvent event = ...;
Iou.Contract contract = Iou.Contract.fromIdAndRecord(event.
↪getContractId(), event.getArguments());
```

(continues on next page)

(continued from previous page)

```
// AFTER
CreatedEvent event = ...;
Iou.Contract contract = Iou.Contract.fromCreatedEvent(event);
```

- Pass the `data.CreatedEvent` directly to the function returned by the decoder's `getDecoder` method. If you are using the decoder utility class method `fromCreatedEvent`, you don't need to change anything.

```
CreatedEvent event = ...;
// BEFORE
Optional<BiFunction<String, Record, Contract>> decoder =
↪MyDecoderUtility.getDecoder(MyTemplate.TEMPLATE_ID);
if (decoder.isPresent()) {
    return decoder.get().apply(event.getContractId(), event.
↪getArguments();
}

// AFTER
Optional<Function<CreatedEvent, Contract>> decoder =
↪MyDecoderUtility.getDecoder(MyTemplate.TEMPLATE_ID);
if (decoder.isPresent()) {
    return decoder.get().apply(event);
}
```

9.2.43.6 Scala Bindings

BREAKING You can now access the *agreement text* of a contract with the new field `Contract#agreementText: Option[String]`.
How to migrate:

- If you are pattern matching on `com.digitalasset.ledger.client.binding.Contract`, you need to add a match clause for the added field.
- If you are constructing `com.digitalasset.ledger.client.binding.Contract` values, for example for tests, you need to add a constructor parameter for the agreement text.

`CreateAndExercise` support via `createAnd` method, e.g. `MyTemplate(owner, someText).createAnd.exerciseAccept(controller, 42)`. See issue #1092 for more information.

9.2.43.7 Ledger

Renamed `--jdbcurl` to `--sql-backend-jdbcurl`. Left `--jdbcurl` in place for backwards compat.
Fixed issue when loading scenarios making use of `pass` into the sandbox, see #1079.
Fixed issue when loading scenarios that involve contract divulgence, see #1166.
Contract visibility is now properly checked when looking up contracts in the SQL backend, see #784.
The sandbox now exposes the *agreement text* of contracts in *CreatedEvents*. See #1110

9.2.43.8 Navigator

Non-empty *agreement texts* are now shown on the contract page above the section `Contract details`, see #1110

9.2.43.9 SQL Extractor

BREAKING In JSON content, dates and timestamps are formatted like `"2020-02-22"` and `"2020-02-22T12:13:14Z"` rather than UNIX epoch offsets like `18314` or `1582373594000000`. See #1174 for more details.

9.2.44 0.12.17 - 2019-05-10

Making transaction lookups performant so we can handle such requests for large ledgers as well
Sandbox: Transactions with a record time that is after the maximum record time (as provided in the original command) are now properly rejected instead of committed to the ledger.
See issue #987 for details.
SDK: The Windows installer no longer requires elevated privileges.

9.2.45 0.12.16 - 2019-05-07

Contract keys: Fixed two issues related to contract key visibility.
See issue #969 and issue #973 for details.
Java Codegen: Variants with unserializable cases are now accepted.
See issue #946 for details.
Java Bindings: `CreateAndExerciseCommand` is now properly converted in the Java Bindings data layer.
See issue #979 for details.
DAML Integration Kit: Alpha release of the kit for integrating your own ledger with DAML. See the DAML Integration Kit docs for how to try it out.
DAML Assistant: Added a `quickstart-scala` DAML Assistant project template.
DAML-LF Engine: If all labels in a record are set, fields no longer need to be ordered.
See issue #988 for details.

9.2.46 0.12.15 - 2019-05-06

Windows support: Beta release of the Windows SDK.
To try it out, download the installer from GitHub releases. The Windows SDK uses the new `daml` command-line which will soon also become the default on Linux and MacOS.
Documentation is still in progress, but you can see the *Migration guide* and the pull request for the updated documentation.
DAML Standard Library: Added `fromListWith` and `merge` to `DA.TextMap`.
DAML Standard Library: Deprecated `DA.Map` and `DA.Set`. Use the new `DA.Next.Map` and `DA.Next.Set` instead.
Ledger API: Added three new methods to the *CommandService*:

- `SubmitAndWaitForTransactionId` returns the transaction ID.

Beta release of the Windows SDK: You can download the installer from GitHub releases. The Windows SDK ships with the new *daml* installer which will soon also become the default on Linux and MacOS. Documentation is still in progress, take a look at the Migration guide and the updated documentation.
Add `fromListWith` and `merge` to `DA.TextMap`.

Release Javadoc artifacts as part of the SDK. See more *here https://github.com/digital-asset/daml/pull/896*
Add `DA.Next.Map` and `DA.Next.Set` and deprecate `DA.Map` and `DA.Set` in favor of those.
Ledger API: Added three new methods to *CommandService*:

- `SubmitAndWaitForTransactionId` returns the transaction id.
- `SubmitAndWaitForTransaction` returns the transaction.
- `SubmitAndWaitForTransactionTree` returns the transaction tree.

Ledger API: Added field `transaction_id` to command completions. This field is only set when a command is successful.
DAML Standard Library: Added instances of `Functor`, `Applicative`, and `Action` for `(->) r` (the reader monad).

9.2.47 0.12.14 - 2019-05-03

DAML Standard Library: The `id` function was previously deprecated and has now been removed. Use `identity` instead.
DAML and Assistant: The compiler no longer supports DAML-LF 1.0.
DAML-LF: As a new dev minor version, writing with `--target 1.dev` is now supported by all tools by default.
Ledger API: You can now look up flat transactions with the new TransactionService methods `GetFlatTransactionByEventId` and `GetFlatTransactionById`.

9.2.48 0.12.13 - 2019-05-02

Sandbox: Fixed an problem with Postgres of potentially not stopping the transaction stream at required ceiling offset.
For more details, see the pull request.

9.2.49 0.12.12 - 2019-04-30

Sandbox: Added support for using a Postgres database as a back end for the Sandbox, which gives you persistent data storage. To try it out, see *DAML Sandbox*.
DAML Integration Kit: Added documentation for the DAML Integration Kit. The docs explain what the DAML Integration Kit is, what state it is in, and how it is going to evolve.
DAML Integration Kit: Released the Ledger API Test Tool. To try it out, see *Ledger API Test Tool*.
DAML-LF: Removed DAML-LF Dev major version, `--target dev` option, and sandbox `--allow-dev` option.
A 1.dev target will handle the intended Dev use cases in a future release.
Ledger API: The list of DAML packages used during interpretation is now included in the produced transaction.
Scala: Source JARs are now released for Scala libraries.
DAML Standard Library: Renamed `DA.TextMap.filter` and `DA.Map.filter` to `filterWithKey`.
Contract keys: Fixed bug releated to visibility and contract keys.
For details, see issue #751.
Contract keys: Fixed bug related witness parties in transaction events.
For details, see issue #794.

9.2.50 0.12.11 - 2019-04-26

Node.js Bindings: The Node.js bindings have been moved to github.com/digital-asset/daml-js.
DAML: Added documentation for flexible controllers. To read about them, see *Overview: template structure*, and for an example, see *Multiple party agreement*.

9.2.51 0.12.10 — 2019-04-25

DAML-LF: DAML-LF 1.3 is now the default compilation target for the DAML compiler. This means that contract keys and text maps are now available by default in DAML.

9.2.52 0.12.9 — 2019-04-23

DAML Standard Library: Added the `DA.Math` library containing exponentiation, logarithms and trig functions
Ledger API: Added `CreateAndExerciseCommand` to the Ledger API and DAMLe for creating a contract and exercising a choice on it within the same transaction.
You can use this to implement callable updates: functions of type `Update a` that can be called from the Ledger API via a contract.
Publish the participant-state APIs and reference implementations.
Sandbox: Added the `-s` option to the CLI to have a shortened version for `--static-time`.
Sandbox: Change `--allow-dev` to be a hidden CLI option, as it's generally not relevant for end users.

9.2.53 0.12.7 — 2019-04-17

No user-facing changes.

9.2.54 0.12.6 — 2019-04-16

Java Bindings: Removed blocking call inside `Bot.wire`, which could lead to an application not making progress in certain situations.

9.2.55 0.12.5 — 2019-04-15

DAML-LF: The DAML-LF Archive Protobuf definitions are now packaged so that it's possible to use them without mangling the path.

9.2.56 0.12.4 — 2019-04-15

SDK: Build artifacts are now released to GitHub.
Sandbox: We now avoid recompiling packages after resetting using the `ResetService`.
Scala: The compiled `google.rpc.Status` is now included in the `ledger-api-scalapb` jar.
Ledger API: Fixed critical bug related to the conversion of decimal numbers from Ledger API.
For details, see issue #399 and issue #439.

9.2.57 0.12.3 — 2019-04-12

SDK: Fix Navigator and Extractor packaging.

9.2.58 0.12.2 — 2019-04-12

DAML: Added flexible controllers and disjunction choices.
Sandbox: Introduced experimental support for using Postgres as a backend. The optional CLI argument for it, `--jdbcurl`, is still hidden.
Node.js Bindings: Fixed validation for Ledger API timestamp values.
Node.js Bindings: Drop support for identifier names, replacing them with separated module and entity names.
Node.js Bindings: Ledger API timestamps and dates are now represented with strings instead of numbers.

Node.js Bindings: Protobuf 64-bit precision integers now use strings instead of numbers, to avoid a loss of precision.
Java Codegen: Added support for DAML TextMap primitive. This is mapped to the `java.util.Map` type, with keys restricted to `java.lang.String` instances.
Java Codegen: Made log output leaner.
Java Codegen: Added flag for log verbosity: `-V LEVEL` or `--verbosity LEVEL`, where `LEVEL` is a number between `0` (least verbose) and `4` (most verbose).
BREAKING - Sandbox and DAMLe: Remove support for DAML 1.0 packages in the engine, and thus the Sandbox. Note that the SDK has removed support for *compiling* DAML 1.0 months ago.

9.2.59 0.12.1 — 2019-04-04

No user-facing changes.

9.2.60 0.12.0 — 2019-04-04

Change in how values are addressed in Navigator's `frontend-config.js`.

- Old syntax for accessing values: `argument.foo.bar`
- New syntax:

```
import { DamlLfValue } from '@da/ui-core';
// Accessing field 'bar' of field 'foo' of the argument
DamlLfValue.evalPath(argument, ["foo", "bar"])
DamlLfValue.toJSON(argument).foo.bar
```

9.2.61 0.11.32

BREAKING CHANGE - DAML standard library: Removed `DA.List.split` function, which was never intended to be exposed and doesn't do what the name suggests.
BREAKING CHANGE - Java Bindings: Removed type parameter for `DamlList` and `DamlOptional` classes.
The `DamlList`, `DamlOptional`, and `ContractId` classes were previously parameterized (i.e `DamlList[String]`) for consistency with the DAML language. The type parameter has been removed as such type information is not supported by the underlying Ledger API and and therefore the parameterized type couldn't be checked for correctness.
BREAKING CHANGE - Java Bindings: For all classes in the package `com.daml.ledger.javaapi.data`, we shortened the names of the conversion methods from long forms like `fromProtoGeneratedCompletionStreamRequest` and `toProtoGeneratedCompletionStreamRequest` to the much shorter `fromProto` and `toProto`.
Navigator: Added support for Optional and recursive data types.
Navigator: Improved start up performance for big DAML models.
BREAKING CHANGE - Navigator: Refactor the GraphQL API.
If you're maintaining a modified version of the Navigator frontend, you'll need to adapt all your GraphQL queries to the new API.
Navigator: Fixed an issue where it was not possible to enter contract arguments involving contract IDs.
Navigator: Fixed issues where the console could not read some events or commands from its database.
BREAKING CHANGE - DAML: For the time being, data types with a single data constructor not associated with an argument are not accepted. For example, `data T = T`.

To work around this, use `data T = T {}` or `data T = T ()` (depending on whether you desire `T` be interpreted as a product or a sum).

9.2.62 0.11.3 - 2019-02-07

Navigator: Fixed display of Date values.
Extractor: Added first version of Extractor with PostgreSQL support.

9.2.63 0.11.2 - 2019-01-31

Navigator: Added a terminal-based console interface using SQLite as a backend.
Navigator: Now writes logs to `./navigator.log` by default using Logback.
DAML Studio: Significant performance improvements.
DAML Studio: New table view for scenario results.
DAML Standard Library: New type classes.
Node.js bindings: Documentation updated to use version 0.4.0 and DAML 1.2.

9.2.64 0.11.1 - 2019-01-24

Java Bindings: Fixed `Timestamp.fromInstant` and `Timestamp.toInstant`.
Java Bindings: Added `Timestamp.getMicroseconds`.

9.2.65 0.11.0 - 2019-01-17

Documentation: *DAML documentation* and *examples* now use DAML 1.2.
Documentation: Added a comprehensive *quickstart guide* that replaces the old My first project example.

As part of this, removed the My first project, IOU and PvP examples.

Documentation: Added a *guide to building applications against a DA ledger*.
Documentation: Updated the *support and feedback page*.
Ledger API: Version 1.4.0 has support for multi-party subscriptions in the transactions and active contracts services.
Ledger API: Version 1.4.0 supports the verbose field in the transactions and active contracts services.
Ledger API: Version 1.4.0 has full support for transaction trees.
Sandbox: Implements Ledger API version 1.4.0.
Java Bindings: Examples updated to use version 2.5.2 which implements Ledger API version 1.4.0.

9.2.65.1 Moving to the new DAML assistant

We've released a new command-line tool for working with the DAML SDK: DAML Assistant, or `daml`. Many of its commands are similar to the old SDK Assistant (`da`), but there are some changes:

Simplified installation process: `curl -sSL https://get.daml.com/ | sh` for Linux and Mac
Overhaul and simplification of templates:

- `daml new` takes arguments in a consistent order:
  - `daml new proj` creates a new project named `proj` with a skeleton template
  - `daml new proj quickstart-java` creates a new project with the quickstart-java template
- `daml new` templates are built-in to the SDK
- Mix-in template mechanism is gone (`da add`)

- No more publishing or subscribing of templates on Bintray: use Github and `git clone` to distribute templates outside of the SDK

Use `daml build` to compile your project into a DAR
`daml start` components don't run in the background, and you stop them with `ctrl+c`
As a result, there are no equivalents to `da stop` and `da restart`
No `da run` equivalent, but:

- `daml sandbox` is the same as `da run sandbox --`
- `daml navigator` is the same as `da run navigator --`
- `daml damlc` is the same as `da run damlc --`

`daml.yaml` configuration file replaces `da.yaml` - read more about this in the next section

Migrating a `da` project to `daml`

Migrating with daml init

You can migrate an existing project using the `daml init` command. To use it, go to the project root on the command line and run `daml init`. This will create a `daml.yaml` file based on `da.yaml`.

Some things to keep in mind when using `daml init` to migrate projects:

If your project uses an SDK version prior to 0.12.15, the generated `daml.yaml` will use SDK version 0.12.15 instead. Support for previous SDK versions in the new assistant is limited.
`daml.yaml` adds `exposed-modules` and `dependencies` fields, which are needed for `daml build`. Depending on your DAML project, may have to adjust these fields in the generated `daml.yaml`.

Migrating manually

To migrate the project manually:

1. Upgrade your project to SDK version 0.12.15 or later.
2. Convert your project's `da.yaml` file into a `daml.yaml` file.
   The two files are very similar: `daml.yaml` is the `project` section of `da.yaml`, plus some additional packaging information. Here is an example of a `daml.yaml` file, from the quickstart-java template:

```
sdk-version: 0.12.14
name: my_project
source: daml
scenario: Main:setup
parties:
- Alice
- Bob
- USD_Bank
- EUR_Bank
version: 1.0.0
exposed-modules:
- Main
dependencies:
- daml-prim
- daml-stdlib
```

   Here is the corresponding `da.yaml` file:

```yaml
project:
  sdk-version: 0.12.12
  scenario: Main:setup
  name: foobar
source: daml
  parties:
  - Alice
  - Bob
  - USD_Bank
  - EUR_Bank
version: 2
```

The extra fields in `daml.yaml` are related to the new packaging functionality in `damlc`. When you build a DAML project with `daml build` (or `daml start`) it creates a `.dar` package from your project inside the `dist/` folder. (You can supply a different target location by passing the `-o` option.) To create the package properly, the new config file `daml.yaml` needs the following additional fields that were not present in `da.yaml`:

`version`: The version number for the DAML project, which becomes the version number for the compiled package.
`exposed-modules`: When the `.dar` file is built, this determines what modules are exposed for users of the package.
`dependencies`: The DAML packages that this project depends on. `daml-prim` and `daml-stdlib` together give access to the basic definitions of DAML - **you should add them both as dependencies**. Additional dependencies can only be added by giving the path to the `.dar` file of the other package.

You can now use `daml` commands with your project.

Switching from old commands to new ones

This section goes through the `da` commands, and gives the `daml` equivalent where there is one.

Managing versions and config

Old command	Purpose	New equivalent
`da setup`	Initialize the SDK	No longer needed: this is handled by the installer
`da upgrade`	Upgrade SDK version	`daml install <version>`
`da list`	List installed SDK versions	`daml version`
`da use`	Set the default SDK version	No direct equivalent; you now set the new SDK version (`sdk-version: X.Y.Z`) in your project config file (`daml.yaml`) manually
`da config`	Query and manage config	No equivalent: view and edit your config files directly
`da uninstall`	Uninstall the SDK	Currently no equivalent for this
`da update-info`	Show assistant update channel information	No longer needed

Running components

Old command	Purpose	New equivalent
`da start`	Start Navigator and Sandbox	`daml start`
`da stop`	Stop running Navigator and Sandbox	`ctrl+c`
`da restart`	Shut down and restart Navigator and Sandbox	`ctrl+c` and `daml start`
`da studio`	Launch DAML Studio	`daml studio`
`da navigator`	Launch Navigator	No direct equivalent; `daml navigator` is equivalent to `da run navigator`
`da sandbox`	Launch Sandbox	No direct equivalent; `daml sandbox` is equivalent to `da run sandbox`
`da compile`	Compile a DAML project into a .dar file	`daml build`
`da run <component>`	Run an SDK component	`daml navigator`, `daml sandbox`, etc as above
`da path <component>`	Show the path to an SDK component	No equivalent
`da status`	Show a list of running services	No longer needed: components no longer run in the background

Managing templates and projects

Old command	Purpose	New equivalent
`da template`	Manage SDK templates	No longer needed: use `git clone` for templates instead
`da project new`	Create an SDK project	`daml new`, or use `git clone`
`da project add`	Add a template to the current project	No longer needed: use `git clone` instead
`da new`	Create a new project from template	`daml new <target path> <name of template>`
`da subscribe`	Subscribe to a template namespace	No longer needed: use `git clone` instead
`da unsubscribe`	Unsubscribe from a template namespace	No longer needed: use `git clone` instead

Docs and feedback

Old command	Purpose	New equivalent
`da docs`	Display the documentation	No longer needed: you can access the docs at docs.daml.com, which includes a PDF download for offline use
`da feedback`	Send us feedback	No longer needed: see *Support* for how to give feedback.
`da config-help`	Show help about config files	No longer needed: config files are documented on this page
`da changelog`	Show release notes	No longer needed: see the *Release notes*

9.3 DAML roadmap (as of September 2019)

This page specifies the major features we're planning to add next to the DAML Ecosystem. Plans and timelines are subject to change. If you need any of these features or want to request others, see the *Support* page for how to get in touch.

We plan to update this roadmap roughly every three months.

DAML Triggers
Support for non-transactional automation written directly in DAML. See discusison on GitHub

Developer tooling
Improved developer experience and functionality on

- Package Management
- Party Allocation and Management
- Application Deployment
- Application Upgrading

API Authentication
Addition of JWT based authentication to the Ledger API

High Level API

A HTTP/JSON-based high-level API with querying capabilities

Ledger Ops Tooling

Tooling for monitoring, logging and health checking ledgers

Canton

Public release of a pre-alpha reference distributed DAML Ledger implementation with a public test-net, strong privacy, regulatory compliance, and composabilty

See canton.io

Deployment Options

- DAML-on-Aurora publicly available on AWS Marketplace
- DAML-on-Sawtooth publicly available on Sextant by Blockchain Technology Partners