DAML SDK Documentation

Digital Asset

Version : 1.5.0-snapshot.20200907.5151.0.eb68e680

Table of contents

Chapter 1

Getting started

1.1 Installing the SDK

1.1.1 1. Install the dependencies

The SDK currently runs on Windows, macOS and Linux.

You need to install:

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

1.1.2 2. Install the SDK

1.1.2.1 Windows 10

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

1.1.2.2 Mac and Linux

To install the SDK on Mac or Linux:

1. In a terminal, run:

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

2. Add `~/.daml/bin` to your PATH. You can find the Mac OS and Linux instructions *here*.

1.1.3 Next steps

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

1.1.4 Alternative: manual download

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

1.1.4.1 Setting JAVA_HOME and PATH variables

Windows

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

Setting the JAVA_HOME variable

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

Setting the PATH variable

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

Mac OS

We'll explain here how to set up JAVA_HOME and PATH variables on Mac OS with zsh shell. If you are using bash all of the instructions are quite similar, except that you will be doing all of the changes in the .bash_profile file.

Setting the JAVA_HOME variable

Run the following command in your terminal:

```
echo 'export JAVA_HOME="$(/usr/libexec/java_home)"' >> ~/.zprofile
```

In order for the changes to take effect you will need to restart your computer. Note that if you will be setting up the PATH variable as well you can restart your computer after you're done with all the changes. Upon restarting check that the JAVA_HOME variable is set:

```
echo $JAVA_HOME
```

The result should be the path to the JDK installation, something like this:

```
/Library/Java/JavaVirtualMachines/jdk_version_number/Contents/Home
```

Setting the PATH variable

Run the following command in your terminal:

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

In order for the changes to take effect you will need to restart your computer. Upon restarting check the PATH variable and make sure that the changes have been applied:

```
echo $PATH
```

You should see ~/.daml/bin in the output.

Linux

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

Setting the JAVA_HOME variable

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

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

In order for the changes to take effect you will need to restart your computer. Note that if you will be setting up the PATH variable as well you can restart your computer after you're done with all the changes. Upon restarting check that the JAVA_HOME variable is set:

```
echo $JAVA_HOME
```

The result should be the path to the JDK installation:

```
/usr/lib/jvm/java-version
```

Setting the PATH variable

Run the following command:

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

Save the file before closing.

In order for the changes to take effect you will need to restart your computer. Upon restarting check the PATH variable and make sure that the changes have been applied:

```
echo $PATH
```

You should see ~/.daml/bin in the output.

1.1.4.2 Manually installing the SDK

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

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

To do that:

1. Go to https://github.com/digital-asset/daml/releases. Confirm your browser sees a valid certificate for the github.com domain.

2. Download the artifact (*Assets* section, after the release notes) for your platform as well as the corresponding signature file. For example, if you are on macOS and want to install release 1.4.0, you would download the files `daml-sdk-1.4.0-macos.tar.gz` and `daml-sdk-1.4.0-macos.tar.gz.asc`. Note that for Windows you can choose between the tarball (ends in `.tar.gz`), which follows the same instructions as the Linux and macOS ones (but assumes you have a number of typical Unix tools installed), or the installer, which ends with `.exe`. Regardless, the steps to verify the signature are the same.
3. To verify the signature, you need to have `gpg` installed (see https://gnupg.org for more information on that) and the Digital Asset Security Public Key imported into your keychain. Once you have `gpg` installed, you can import the key by running:

   ```
   gpg --keyserver pool.sks-keyservers.net --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 1.4.0 on macOS, you should have both files in the current directory and run:

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

   and that should give you a result that looks like:

   ```
   gpg: assuming signed data in 'daml-sdk-1.4.0-macos.tar.gz'
   gpg: Signature made Wed Aug 12 13:30:49 2020 CEST
   gpg:                using RSA key E8372C0C1C734C51
   gpg: Good signature from "Digital Asset Holdings, LLC
   ↪<security@digitalasset.com>" [unknown]
   gpg: WARNING: This key is not certified with a trusted signature!
   gpg:          There is no indication that the signature belongs to the
   ↪owner.
   Primary key fingerprint: 4911 A8DF E976 ACDF A071  30DB E837 2C0C 1C73
   ↪4C51
   ```

   Note: This warning means you have not told gnupg that you trust this key actually belongs to Digital Asset. The `[unknown]` tag next to the key has the same meaning: `gpg` relies on a web of trust, and you have not told it how far you trust this key. Nevertheless, at this point you have verified that this is indeed the key that has been used to sign the archive.
5. The next step is to extract the tarball and run the install script (unless you chose the Windows installer, in which case the next step is to double-click it):

   ```
   tar xzf daml-sdl-1.4.0-macos.tar.gz
   cd sdk-1.4.0
   ./install.sh
   ```

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

1.2 Getting Started with DAML

The goal of this tutorial is to get you up and running with full-stack DAML development. We do this through the example of a simple social networking application, showing you three things:

1. How to build and run the application
2. The design of its different components (app-architecture)
3. How to write a new feature for the app (first-feature)

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

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

With that, let's get started!

1.2.1 Prerequisites

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

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

- Git version control system
- Yarn package manager for JavaScript. You have to have yarn version 1.10.0 or higher.
  Note: Ubuntu 17.04 and higher come with `cmdtest` package installed by default. If you are getting errors when installing yarn, you may want to run `sudo apt remove cmdtest` first and then install yarn. More information can be found here as well as in the official yarn installation docs for Debian / Ubuntu
- NodeJS in version 8.16 or higher. This will usually be installed automatically as part of installing Yarn.
  Note: On Ubuntu 18.04, NodeJS 8.10 will be installed as part of installing Yarn which is too old. You can find instructions for installing newer versions at NodeSource.
- A terminal application for command line interaction

1.2.2 Running the app

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

First off, open a terminal and instantiate the template project.

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

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

Change to the new folder:

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

Next we need to compile the DAML code to a DAR file:

```
daml build
```

Once the DAR file is created you will see this message in terminal `Created .daml/dist/create-daml-app-0.1.0.dar`.

Any commands starting with `daml` are using the *DAML Assistant*, a command line tool in the DAML SDK for building and running DAML apps. In order to connect the UI code to this DAML, we need to run a code generation step:

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

Now, changing to the `ui` folder, use Yarn to install the project dependencies:

```
cd ui
yarn install --force --frozen-lockfile
```

This step may take a couple of moments (it's worth it!). You should see `success Saved lockfile.` in the output if everything worked as expected.

We can now run the app in two steps. You'll need two terminal windows running for this. In one terminal, at the root of the `create-daml-app` directory, run the command:

```
daml start
```

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

1. Compiles the DAML code to a DAR file as in the previous `daml build` step.
2. Starts an instance of the *Sandbox*, an in-memory ledger useful for development, loaded with our DAR.
3. Starts a server for the *HTTP JSON API*, a simple way to run commands against a DAML ledger (in this case the running Sandbox).

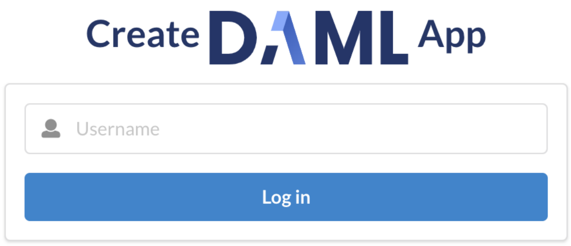
We'll leave these processes running to serve requests from our UI.

In a second terminal, navigate to the `create-daml-app/ui` folder and run the application:

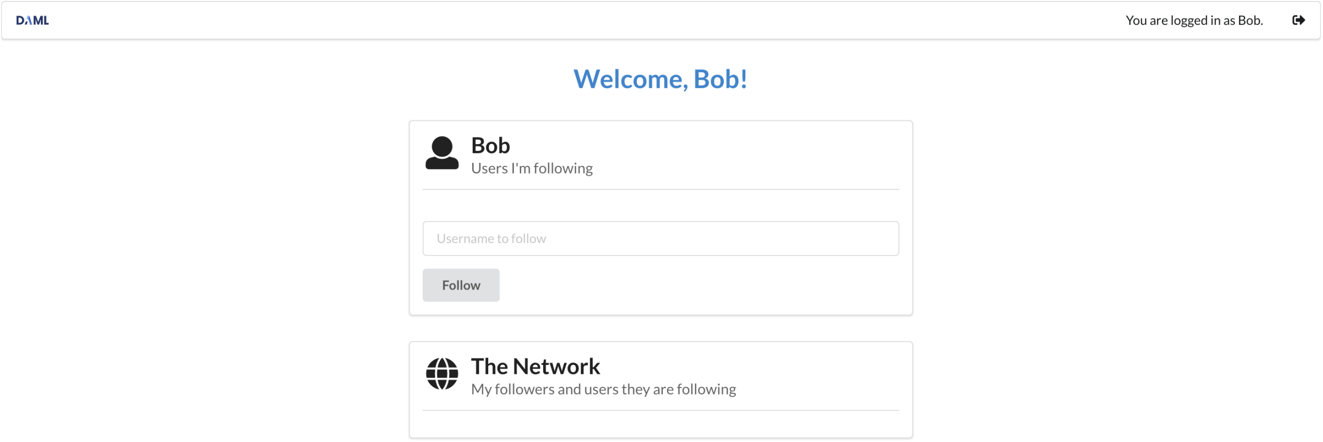
```
cd ui
yarn start
```

This starts the web UI connected to the running Sandbox and JSON API server. The command should automatically open a window in your default browser at http://localhost:3000. Once the web UI has been compiled and started, you should see `Compiled successfully!` in your terminal. If it doesn't, just open that link in a web browser. (Depending on your firewall settings, you may be asked whether to allow the app to receive network connections. It is safe to accept.) You should now see the login page for the social network. For simplicity of this app, there is no password or sign-up required. First enter your name and click *Log in*.

You should see the main screen with two panels. One for the users you are following and one for your followers. Initially these are both empty as you are not following anyone and you don't have any followers! Go ahead and start following users by typing their usernames in the text box and clicking on the *Follow* button in the top panel.



Create DAML App



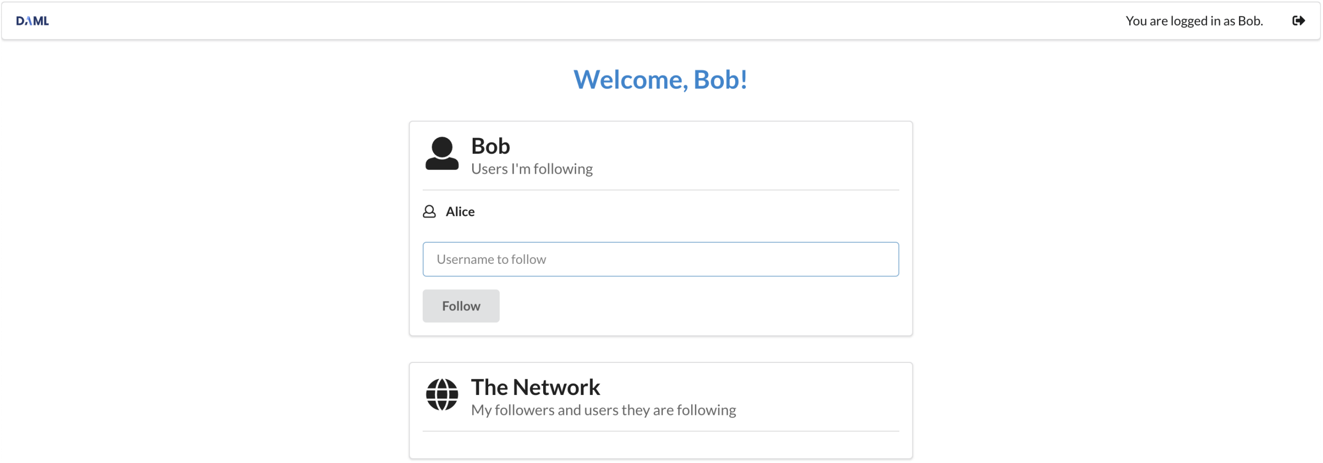
You are logged in as Bob.

Welcome, Bob!

Bob
Users I'm following

The Network
My followers and users they are following

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



You are logged in as Bob.

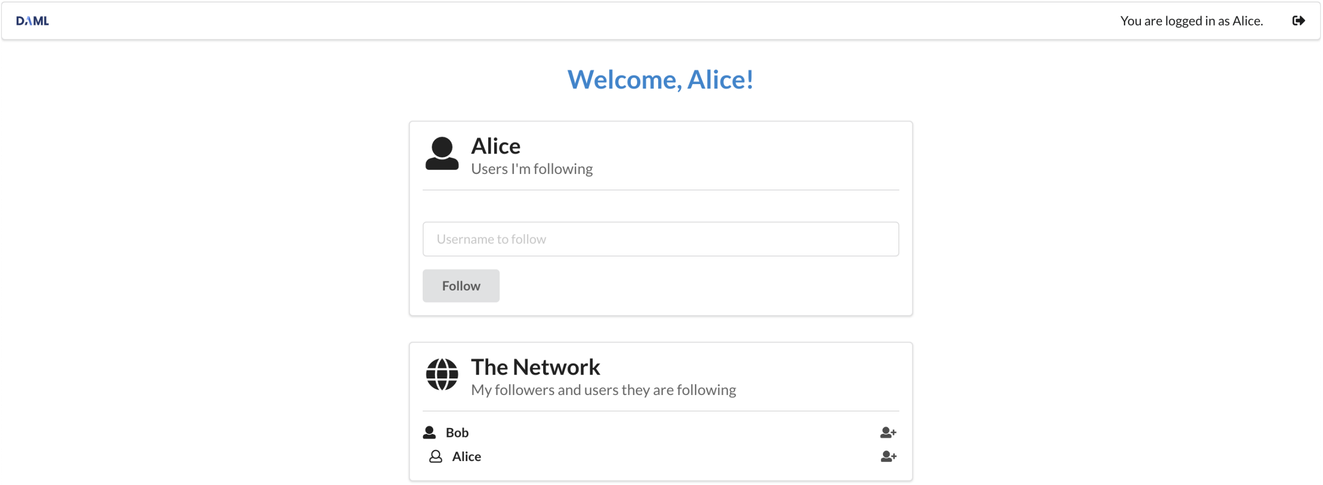
Welcome, Bob!

Bob
Users I'm following

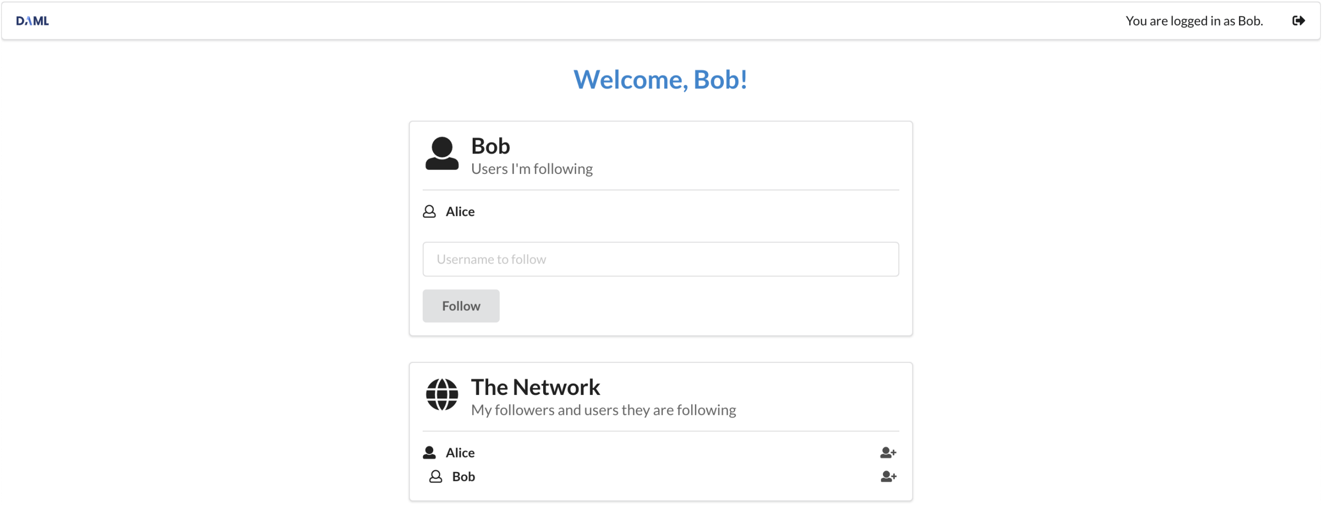
Alice

The Network
My followers and users they are following

To make this relationship reciprocal, open a new browser window/tab at http://localhost:3000. (Having separate windows/tabs allows you to see both you and the screen of the user you are following at the same time.) Once you log in as the user you are following - Alice, you'll notice your name in her network. In fact, Alice can see the entire list of users you are follwing in the *Network* panel. This is because this list is part of the user data that became visible when you started follwing her.



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



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

1.3 Testing Your App

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

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

> Jest is a general-purpose testing framework for JavaScript that's well integrated with both TypeScript and React. Jest helps you structure your tests and express expectations of the app's

behaviour.

Puppeteer is a library for controlling a Chrome browser from JavaScript/TypeScript. Puppeteer allows you to simulate interactions with the app in place of a real user.

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

```
yarn add --dev puppeteer wait-on @types/jest @types/node @types/puppeteer
 ↪@types/wait-on
```

1.3.1 Setting up our tests

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

```
yarn test
```

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

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

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

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

1.3.2 Example: Logging in and out

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

We'll walk though this step by step.

The `test` syntax is provided by Jest to indicate a new test running the function given as an argument (along with a description and time limit).

`getParty()` gives us a new party name. Right now it is just a string unique to this set of tests, but in the future we will use the Party Management Service to allocate parties.

`newUiPage()` is a helper function that uses the Puppeteer browser to open a new page (we use one page per party in these tests), navigate to the app URL and return a `Page` object.

Next we `login()` using the new page and party name. This should take the user to the main screen. We'll show how the `login()` function does this shortly.

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

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

1.3.3 Accessing UI elements

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

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

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

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

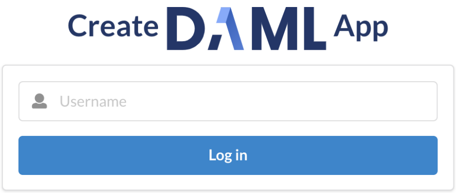
You can see the `className` attributes in the `Input` and `Button`, which we select in the `login()` function. Note that you can use other features of an element in your selector, such as its type and attributes. We've only used class selectors in these tests.

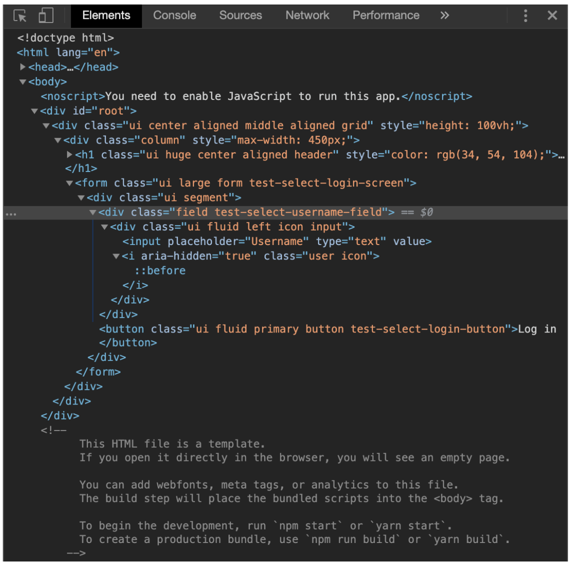
1.3.4 Writing CSS Selectors

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

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

1.3.5 The Full Test Suite





```
<!doctype html>
<html lang="en">
 ▶<head>…</head>
 ▼<body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
   ▼<div id="root">
     ▼<div class="ui center aligned middle aligned grid" style="height: 100vh;">
       ▼<div class="column" style="max-width: 450px;">
         ▶<h1 class="ui huge center aligned header" style="color: rgb(34, 54, 104);">…
          </h1>
         ▼<form class="ui large form test-select-login-screen">
           ▼<div class="ui segment">
...          ▼<div class="field test-select-username-field"> == $0
               ▼<div class="ui fluid left icon input">
                  <input placeholder="Username" type="text" value>
                 ▼<i aria-hidden="true" class="user icon">
                    ::before
                  </i>
                </div>
              </div>
              <button class="ui fluid primary button test-select-login-button">Log in
              </button>
            </div>
          </form>
        </div>
      </div>
    </div>
    <!--
          This HTML file is a template.
          If you open it directly in the browser, you will see an empty page.

          You can add webfonts, meta tags, or analytics to this file.
          The build step will place the bundled scripts into the <body> tag.

          To begin the development, run `npm start` or `yarn start`.
          To create a production bundle, use `npm run build` or `yarn build`.
      -->
```

Chapter 2

Writing DAML

2.1 An introduction to DAML

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

In this introduction, you will learn about the structure of a DAML Ledger, and how to write DAML applications that run on any DAML Ledger implementation, by building an asset-holding and -trading application. You will gain an overview over most important language features, how they relate to the *DAML Ledger Model* and how to use 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 load the sources for section 1 into a folder called `1_Token`, run `daml new 1_Token --template daml-intro-1`.

Prerequisites:

You have installed the DAML SDK

Next: *1 Basic contracts*.

2.1.1 1 Basic contracts

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

Transactions
DAML Modules and Files
Templates
Contracts
Signatories

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

2.1.1.1 DAML ledger basics

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

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

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

```
module Token where
```

Code comments in DAML are introduced with –:

```
-- The first line of a DAML file is a pragma telling the
↪language
-- version to use.

-- 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

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

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.

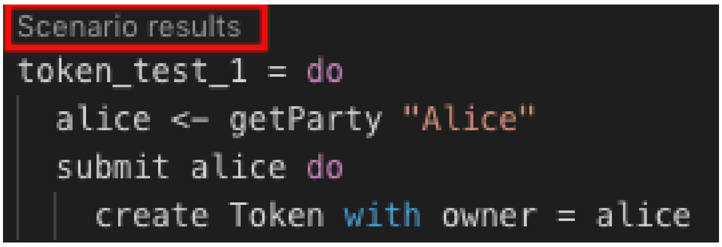
2.1.2.2 Running scenarios

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.



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

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.

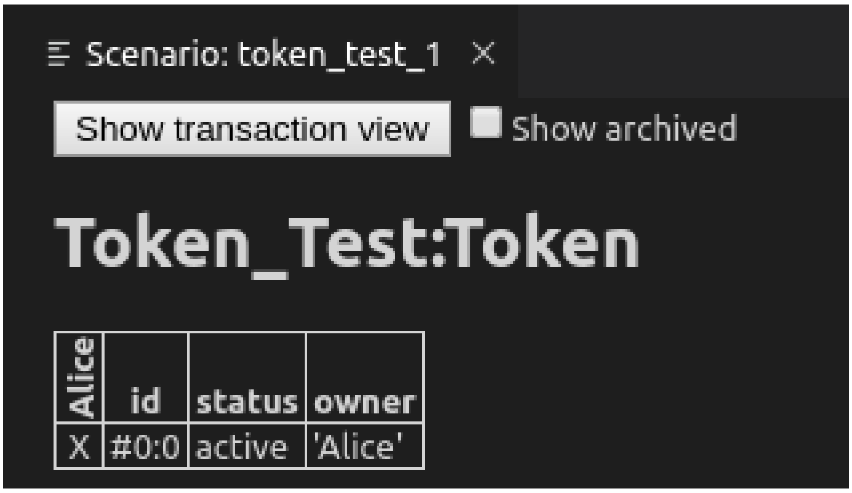
The first columns, labelled vertically, show which parties know about which contracts. In this simple scenario, the sole party Alice knows about the contract she created.

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.



Token_Test:Token

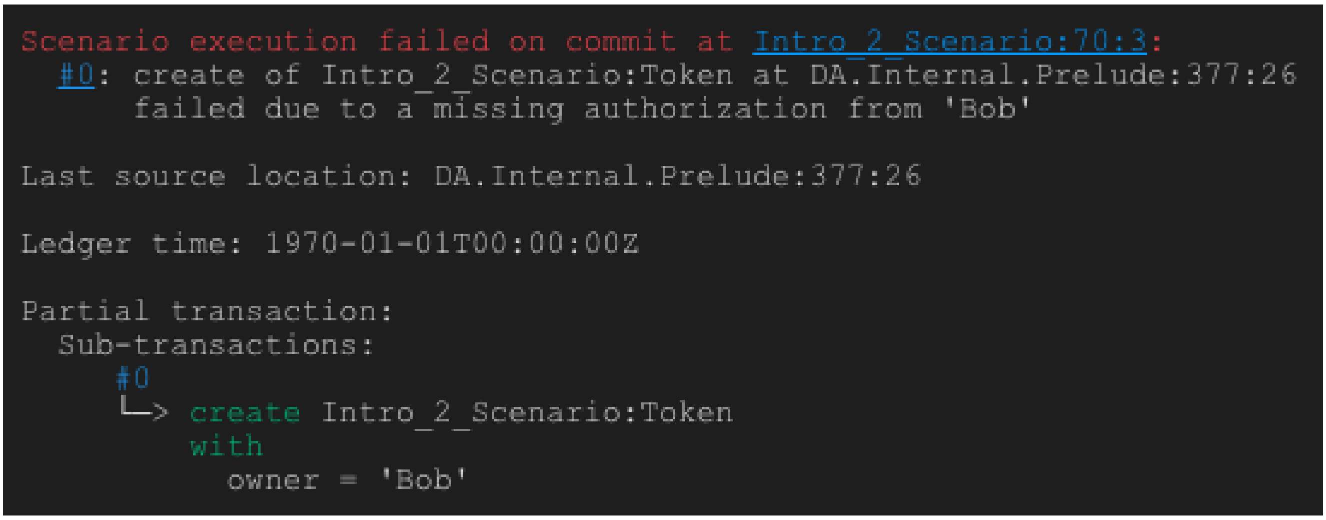
Alice	id	status	owner
X	#0:0	active	'Alice'

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

  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.

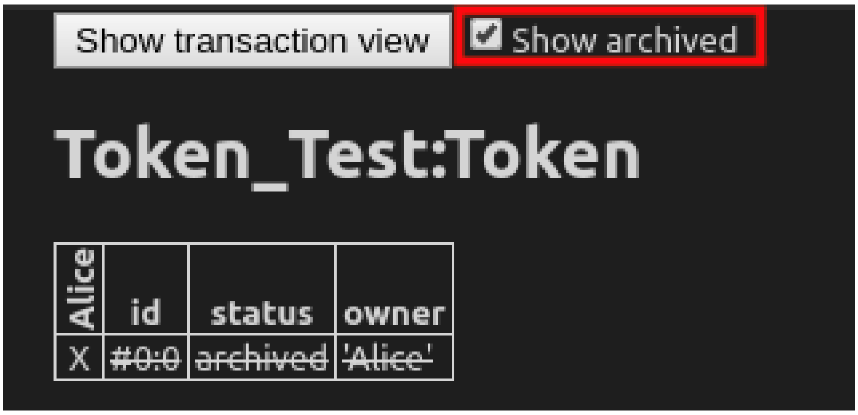
To archive the Token Alice has created, you need to get a handle on its contract ID. In scenarios, you do this using <- notation. That's because the contract ID needs to be retrieved from the ledger. How this works is discussed in *5 Adding constraints to a contract*.

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:

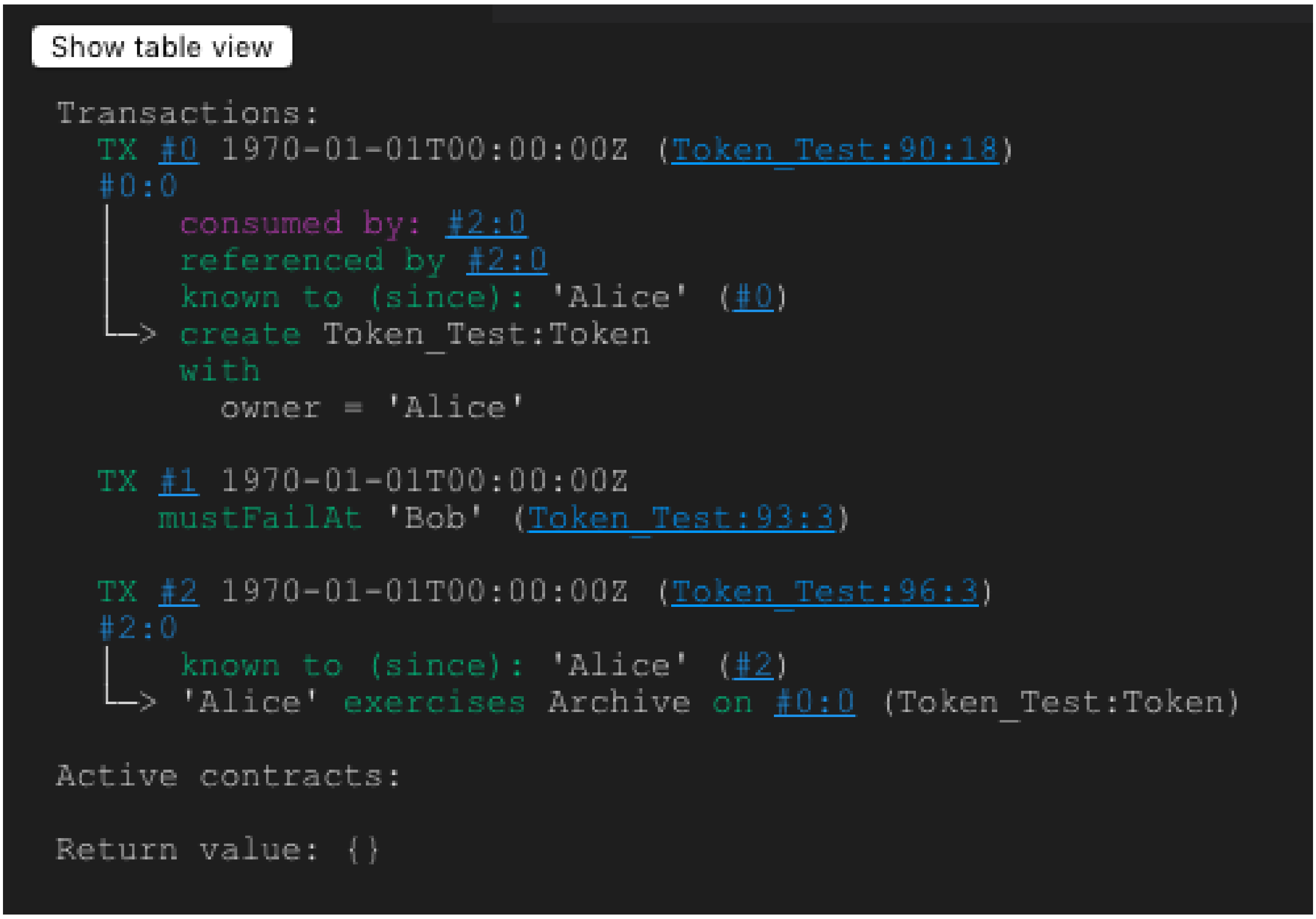


Token_Test:Token

Alice	id	status	owner
X	~~#0:0~~	~~archived~~	~~'Alice'~~

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:



```
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: {}
```

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.

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.

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

2.1.3.1 Native types

You have already encountered a few native DAML types: `Party` in *1 Basic contracts*, and `Text` and `ContractId` in *2 Testing templates using 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 opaque.
> `Text` Stores a unicode character string like `"Alice"`.
> `ContractId a` Stores a reference to a contract of type `a`.
> `Int` Stores signed 64-bit integers. For example, `-123`.
> `Decimal` Stores fixed-point number with 28 digits before and 10 digits after the decimal point. For example, `0.0000000001` or `-9999999999999999999999999999.9999999999`.
> `Bool` Stores `True` or `False`.
> `Date` Stores a date.
> `Time` Stores absolute UTC time.
> `RelTime` Stores a difference in time.

The below 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 : Decimal
    my_text = "Alice"
    my_bool = False
    my_date = date 2020 Jan 01
    my_time = time my_date 00 00 00
    my_rel_time = hours 24

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

Despite its simplicity, there are quite a few things to note in this 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.
Most variables do not have annotations to say what type they are.
That's because DAML is very good at *inferring* types. The compiler knows that `123` is an `Int`, so if you declare `my_int = 123`, it can infer that `my_int` is also an `Int`. This means you don't have to write the type annotation `my_int : Int = 123`.
However, if the type is ambiguous so that the compiler can't infer it, you do have to add a type annotation. This is the case for `0.001` which could be any `Numeric n`. Here we specify `0.001 : Decimal` which is a synonym for `Numeric 10`. You can always choose to add type annotations to aid readability.
The `assert` function is an action that takes a boolean value and succeeds with `True` and fails with `False`.
Try putting `assert False` somewhere in a 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"
```

(continues on next page)

(continued from previous page)

```
      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 : Decimal, 2.0 : Decimal, 3.0 : Decimal)

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

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

You can access the data in the tuples using:

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

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

Lists

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

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

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

```
import DA.List

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

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

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

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

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

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

Records

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

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

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

-- 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
```

(continues on next page)

(continued from previous page)

```
  my_record : MyRecord
  my_kv : KeyValue Text Int

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

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

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

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

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

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

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

In the `assert` statements above, we always compared values of in-built types. If you wrote `assert (my_record == my_record)` in the 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
  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
```

(continues on next page)

(continued from previous page)

```
  bank : Bank
    deriving (Eq, Show)

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

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

cash_balance_test = 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
    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
```

(continues on next page)

(continued from previous page)

```
      cash
      location = InHand

  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
```

(continues on next page)

(continued from previous page)

```
    r : Either Int Text = Right "r"

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

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

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

    day = Sunday
    weekend = case day of
      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
```

(continues on next page)

(continued from previous page)

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

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

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

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

id_ref_test = 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
```

(continues on next page)

(continued from previous page)

```
      accountant
      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 maintainers are part of the key is ensured by only having the variable *key* in each maintainer expression.

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.

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

2.1.4.1 Choices as methods

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

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

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

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

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

Let's unpack the code snippet above:

The first line, `controller owner can` says that the following choices are *controlled* by `owner`, meaning `owner` is the only party that is allowed to *exercise* them. The line starts a new block in which multiple choices can be defined.

`UpdateTelephone` is the name of a choice. It starts a new block in which that choice is defined.

`: ContractId Contact` is the return type of the choice.

This particular choice archives the current `Contact`, and creates a new one. What it returns is a reference to the new contract, in the form of a `ContractId Contact`

The following `with` block is that of a record. Just like with templates, in the background, a new record type is declared: `data UpdateTelephone = UpdateTelephone with`

The `do` starts a block defining the action the choice should perform when exercised. In this case a new `Contact` is created.

The new `Contact` is created using `this with`. `this` is a special value available within the `where` block of templates and takes the value of the current contract's arguments.

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

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 convenience. Choices *always* return an Update so for readability it's omitted on the type declaration of a choice.

Now to exercise the new choice in a 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"

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

  submit owner do
    newContact <- fetch newContactCid
    assert (newContact.telephone == "098 7654 321")
```

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

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

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

  submit owner do
    newContact <- fetch newContactCid
    assert (newContact.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 four kinds of action: `create`, `exercise`, `fetch` and `key assertion`.

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

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

(continues on next page)

(continued from previous page)

```
        owner = 'Alice'; party = 'Bob'; address = "1 Bobstreet"; telephone
↪= "012 345 6789"

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

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

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

  TX #4 1970-01-01T00:00:00Z (Contact:66:22)
  #4:0
  │   known to (since): 'Alice' (#4), 'Bob' (#4)
  └─> 'Bob' exercises UpdateAddress on #2:1 (Contact:Contact)
            with
              newAddress = "1-10 Bobstreet"
      children:
      #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
```

(continues on next page)

(continued from previous page)

```
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) 2020 Digital Asset (Switzerland) GmbH and/or its affiliates. All rights reserved.
-- SPDX-License-Identifier: Apache-2.0


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
```

(continues on next page)

(continued from previous page)

```
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
    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.

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

2.1.5.1 Template preconditions

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

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

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

    ensure cash.amount > 0.0
```

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

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

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

```
test_restrictions = 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
```

(continues on next page)

(continued from previous page)

```
      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.
  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.
```

(continues on next page)

(continued from previous page)

```
  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
            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 at 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 time (LT)* and the *record time (RT)*. The ledger time is set by the participant, the record time is set by the ledger.

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

`getTime` is an action that gets the LT from the ledger. In the above example, that time is taken apart into day of week and hour of day using standard library functions from `DA.Date` and `DA.Time`. The hour of the day is checked to be in the range from 8 to 18.

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

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

For details, see Background concepts - time.

Time in scenarios

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

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

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

pass (days (-3))
```

(continues on next page)

(continued from previous page)

```
  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 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 build complex actions from simple ones, 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"
```

(continues on next page)

(continued from previous page)

```
  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
    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 random number generator's 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 Lipova a)
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 base exponent
 | exponent == 0 = Some 1
 | exponent > 0 =
   let Some result = optPow base (exponent - 1)
   in Some (base * result)
 | otherwise = None
```

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

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

The big downside to this is that even unused errors cause failures. The following 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

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

2.1.6.1 Preventing IOU revocation

The `SimpleIou` contract from *4 Transforming data using choices* and *5 Adding constraints to a contract* has one major problem: The contract is only signed by the `issuer`. The signatories are the parties with the power to create and archive contracts. If Alice gave Bob a `SimpleIou` for $100 in exchange for some goods, she could just archive it again after receiving the goods. Bob would have a record of such actions, but would have to resort to off-ledger means to get his money back.

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

```
simple_iou_test = 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
```

(continues on next page)

(continued from previous page)

```
        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

    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 Bob's authority.
  submitMustFail alice do
    create Iou with
      issuer = alice
      owner = bob
      cash = Cash with
        amount = 100.0
        currency = "USD"
```

(continues on next page)

(continued from previous page)

```
  -- 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"

  -- However, she 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 building issuance and transfer workflows for the above `Iou`, before diving into the authorization model in full.

2.1.6.2 Use propose-accept workflows for one-off authorization

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

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

    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"
```

(continues on next page)

(continued from previous page)

```
  submit bob do
    exercise iouProposal IouProposal_Accept
```

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

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

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

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

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

    controller newOwner can
      IouTransferProposal_Reject
        : ContractId Iou
        do
          create iou

      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
        amount = 100.0
        currency = "USD"

-- Bob accepts the transfer from Alice.
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 the transfer from Bob.
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 scenario 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
```

(continues on next page)

(continued from previous page)

```
   with
    newOwner : Party
   controller owner, newOwner
   do
    create this with
     owner = newOwner
```

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

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

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

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

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

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

Here it is in action:

```
 -- Bob allows Alice to send him Ious.
 sab <- submit bob do
  create IouSender with
   sender = alice
   receiver = bob

```

(continues on next page)

(continued from previous page)

```
-- 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 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:
    #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
```

(continues on next page)

(continued from previous page)

```
    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
  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 *IOU Quickstart Tutorial*. In that context you'll learn a bit more about the `Update` action and how to use it to compose transactions, as well as about privacy on DAML ledgers.

2.1.7 7 Composing choices

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

DAML projects, packages and modules

Composition of transactions
Observers and stakeholders
DAML's execution model
Privacy

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

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

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 a `daml.yaml` file and an empty directory of source files. Take a look at the `daml.yaml` for this project:

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

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 `.daml/dist/dist/project_name-project_version.dar`. A `dar` file is DAML's equivalent of a `JAR` file in Java: it's the artifact that gets deployed to a ledger to load the 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
│   └── 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:

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

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

```
import Intro.Asset
```

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

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

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 *6 Parties and authority* to good use. For example, the Merge and Split choices each perform several actions in their consequences.

Two create actions in case of Split

One create and one archive action in case of Merge

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

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

(continues on next page)

(continued from previous page)

```
          fetchedQuoteAsset <- fetch quoteAssetCid
          assertMsg
            "Quote asset mismatch"
            (quoteAsset == fetchedQuoteAsset with
              observers = quoteAsset.observers)

          transferredBaseCid <- exercise
            baseApprovalCid TransferApproval_Transfer with
              assetCid = baseAssetCid

          transferredQuoteCid <- exercise
            quoteApprovalCid TransferApproval_Transfer with
              assetCid = quoteAssetCid

          return (transferredBaseCid, transferredQuoteCid)
```

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

```
TX #15 1970-01-01T00:00:02Z (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)
```

(continues on next page)

(continued from previous page)

```
        └─> 'Alice', 'USD_Bank' exercises Archive on #11:1 (Intro.
↪Asset:Asset)

        #15:6
        │   referenced by #17:0
        │   known to (since): 'Bob' (#15), 'USD_Bank' (#15), 'Alice' (#15)
        └─> create Intro.Asset:Asset
            with
              issuer = 'USD_Bank'; owner = 'Bob'; symbol = "USD"; quantity
↪= 100.0; observers = []

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

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

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

Similar to choices, you can see how the 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
```

(continues on next page)

(continued from previous page)

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

2.1.7.6 Observers

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

```
template Asset
  with
    issuer : Party
    owner : Party
    symbol : Text
    quantity : Decimal
    observers : [Party]
  where
    signatory issuer, owner
    ensure quantity > 0.0

    observer observers
```

The `Asset` template also gives the `owner` a choice to set the observers, and you can see how Alice uses it to show her `Asset` to Bob just before proposing the trade. You can try out what happens if she didn't do that by removing that transaction.

```
  usdCid <- submit alice do
    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 counterparties, she would otherwise end up leaking them to each other.

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

2.1.7.7 Privacy

DAML's privacy model is based on two principles:

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 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`.

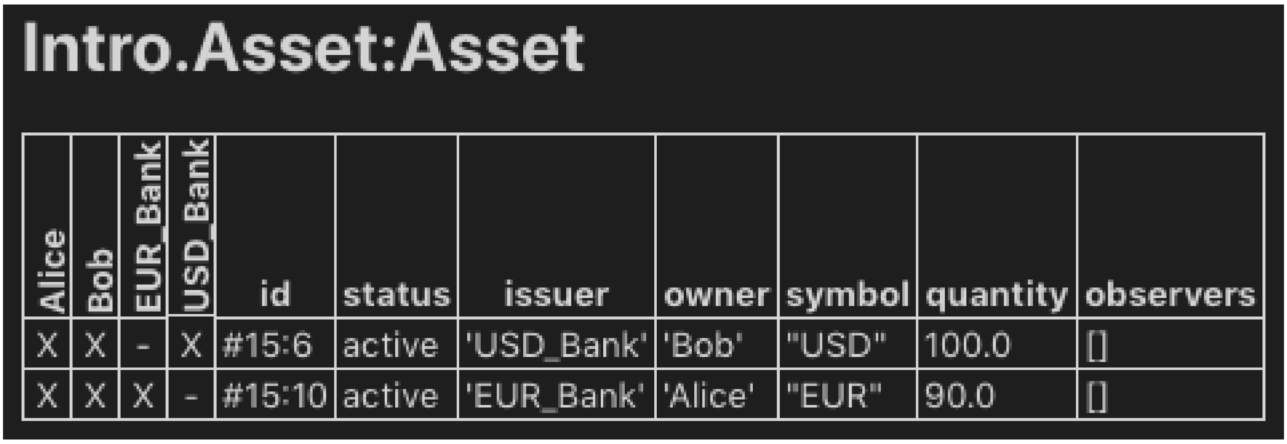
Each of the two involved `TransferApproval` contracts is signed by a different `issuer`, which see the action on their contract. So the EUR_Bank sees the `TransferApproval_Transfer` action for the EUR `Asset` and the USD_Bank sees the `TransferApproval_Transfer` action for the USD `Asset`.

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 disclosure 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
    -- more parameters here
  where
    signatory exampleParty
    observer exampleParty2
    agreement
      -- some text
      ""
    ensure
      -- boolean condition
      True
    key (exampleParty, exampleParameter) : (Party, Text)
    maintainer (exampleFunction key)
    -- a choice goes here; see next section
```

template name `template` keyword

parameters `with` followed by the names of parameters and their types

template body `where` keyword

Can include:

- ***template-local definitions*** `let` keyword

  Lets you make definitions that have access to the contract arguments and are available in the rest of the template definition.

- ***signatories*** `signatory` keyword

  Required. The parties (see the *Party* type) who must consent to the creation of 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
    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.

consumption annotation Optionally one of `preconsuming`, `postconsuming`, `nonconsuming`, which changes the behavior of the choice with respect to privacy and if and when the contract is archived. See *contract consumption in choices* for more details.

a name Must begin with a capital letter. Must be unique - choices in different templates can't have the same name.

a return type after a `:`, the return type of the choice

choice arguments `with` keyword
If you start your choice with `choice` and include a `Party` as a parameter, you can make that `Party` the `controller` of the choice. This is a feature called flexible controllers , and it means you don't have to specify the controller when you create the contract - you can specify it when you exercise the choice. To exercise a choice, the party needs to be a signatory or an observer of the contract and must be explicitly declared as such.

a choice body After `do` keyword
What happens when someone exercises the choice. A choice body can contain update statements: see *Choice body structure* below.

2.2.1.3 Choice body structure

A choice body contains `Update` expressions, wrapped in a *do* block.

The update expressions are:

create Create a new contract 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 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
```

(continues on next page)

(continued from previous page)

```
    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 Template-local Definitions

```
  where
    let
      allParties = [exampleParty, exampleParty2, exampleParty3]
```

let keyword. Starts a block and is followed by any number of definitions, just like any other let block.
Template parameters as well as this are in scope, but self is not.
Definitions from the let block can be used anywhere else in the template's where block.

2.2.2.4 Signatory parties

```
    signatory exampleParty
```

signatory keyword. After where. Followed by at least one Party.
Signatories are the parties (see the Party type) who must consent to the creation of 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.
Each template **must** have at least one signatory. A signatory declaration consists of the *signatory* keyword followed by a comma-separated list of one or more expressions, each expression denoting a Party or collection thereof.

2.2.2.5 Observers

```
    observer exampleParty2
```

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

2.2.2.6 Choices

```
-- option 1 for specifying choices: choice name first
choice NameOfChoice1
      : ()  -- replace () with the actual return type
    with
      exampleParameter : Text -- parameters here
  controller exampleParty
    do
      return () -- replace this line with the choice body

-- option 2 for specifying choices: controller first
controller exampleParty can
  NameOfChoice2
      : () -- replace () with the actual return type
    with
      exampleParameter : Text -- parameters here
    do
      return () -- replace this line with the choice body
  nonconsuming NameOfChoice3
      : ()  -- replace () with the actual return type
    with
      exampleParameter : Text -- parameters here
    do
      return () -- replace this line with the choice body
```

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

2.2.2.7 Agreements

```
agreement
  -- text representing the contract
  ""
```

`agreement` keyword, followed by text.

Represents what the contract means in text. They're usually the boundary between on-ledger and off-ledger rights and obligations.
Usually, they look like `agreement tx`, where `tx` is of type `Text`.
You can use the built-in operator `show` to convert party names to a string, and concatenate with `<>` .

2.2.2.8 Preconditions

```
ensure
  True -- a boolean condition goes here
```

`ensure` keyword, followed by a boolean condition.
Used on contract creation. `ensure` limits the values on parameters that can be passed to the contract: the contract can only be created if the boolean condition is true.

2.2.2.9 Contract keys and maintainers

```
key (exampleParty, exampleParam) : (Party, Text)
maintainer (exampleFunction key)
```

`key` and `maintainer` keywords.
This feature lets you specify a key that you can use to uniquely identify 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:

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.

Contract consumption

If no qualifier is present, choices are *consuming*: the contract is archived before the evaluation of the choice body and both the controllers and all contract stakeholders see all consequences of the action.

2.2.3.4 Preconsuming choices

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

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

Listing 7: Option 2 for specifying choices: controller first

```
      preconsuming ExampleChoice7
          : () -- replace () with the actual return type
```

`preconsuming` keyword. Optional.
Makes a choice pre-consuming: the contract is archived before the body of the exercise is executed.
The archival behavior is analogous to the *consuming* default behavior.
Unlike what happens the in *consuming* behavior, though, only the controllers and signatories of the contract see all consequences of the action. If the choice archives the contract, other stakeholders merely see an archive action.
Can be thought as a non-consuming choice that implicitly archives the contract before anything else happens

2.2.3.5 Postconsuming choices

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

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

Listing 9: Option 2 for specifying choices: controller first

```
      postconsuming ExampleChoice8
          : () -- replace () with the actual return type
```

`postconsuming` keyword. Optional.
Makes a choice post-consuming: the contract is archived after the body of the exercise is executed.
The contract can still be used in the body of the exercise.
Only the controllers and signatories of the contract see all consequences of the action. If the choice archives the contract, other stakeholders merely see an archive action.
Can be thought as a non-consuming choice that implicitly archives the contract after the choice has been exercised

2.2.3.6 Non-consuming choices

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

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

Listing 11: Option 2 for specifying choices: controller first

```
      nonconsuming ExampleChoice4
          : () -- replace () with the actual return type
```

`nonconsuming` keyword. Optional.
Makes a choice non-consuming: that is, exercising the choice does not archive the contract.
Only the controllers and signatories of the contract see all consequences of the action. If the choice archives the contract, other stakeholders merely see an archive action.
Useful in the many situations when you want to be able to exercise a choice more than once.

Return type

Return type is written immediately after choice name.
All choices have a return type. A contract returning nothing should be marked as returning a unit , ie `()`.
If a contract is/contracts are created in the choice body, usually you would return the contract ID(s) (which have the type `ContractId <name of template>`). This is returned when the choice is exercised, and can be used in a variety of ways.

2.2.3.7 Choice arguments

```
        with
          exampleParameter : Text
```

`with` keyword.
Choice arguments are similar in structure to *Template parameters*: a *record type*.
A choice argument can't have the same name as any *parameter to the template* the choice is in.
Optional - only if you need extra information passed in to exercise the choice.

2.2.3.8 Choice body

Introduced with `do`
The logic in this section is what is executed when the choice gets exercised.
The choice body contains `Update` expressions. For detail on this, see *Reference: updates*.
By default, the last expression in the choice is returned. You can return multiple updates in tuple form or in a custom data type. To return something that isn't of type `Update`, use the `return` keyword.

2.2.4 Reference: updates

This page gives reference information on Updates:

For the structure around them, see *Overview: template structure*.

2.2.4.1 Background

An `Update` is ledger update. There are many different kinds of these, and they're listed below. They are what can go in a *choice body*.

2.2.4.2 Binding variables

```
boundVariable <- UpdateExpression1
```

One of the things you can do in a choice body is bind (assign) an Update expression to a variable. This works for any of the Updates below.

2.2.4.3 do

```
do
   updateExpression1
   updateExpression2
```

`do` can be used to group `Update` expressions. You can only have one update expression in a choice, so any choice beyond the very simple will use a `do` block.
Anything you can put into a choice body, you can put into a `do` block.

By default, `do` returns whatever is returned by the **last expression in the block**.
So if you want to return something else, you'll need to use `return` explicitly - see *return* for an example.

2.2.4.4 create

```
create NameOfTemplate with exampleParameters
```

`create` function.
Creates 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.
Like `fetch`, `fetchByKey` needs to be authorized by at least one stakeholder of the contract.
Fails if no contract can be found.

2.2.4.9 lookupByKey

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

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

2.2.4.10 abort

```
abort errorMessage
```

`abort` function.
Fails the transaction - nothing in it will be committed to the ledger.
`errorMessage` is of type `Text`. Use the error message to provide more context to an external system (e.g., it gets displayed in DAML Studio scenario results).
You could use `assert False` as an alternative.

2.2.4.11 assert

```
assert (condition == True)
```

`assert` keyword.
Fails the transaction if the condition is false. So the choice can only be exercised if the boolean expression evaluates to `True`.
Often used to restrict the arguments that can be supplied to a contract choice.

Here's an example of using `assert` to prevent a choice being exercised if the `Party` passed as a parameter is on a blacklist:

```
    Transfer : ContractId RestrictedPayout
      with newReceiver : Party
      do
```

(continues on next page)

(continued from previous page)

```
          assert (newReceiver /= blacklisted)
          create RestrictedPayout with receiver = newReceiver; giver;
↪blacklisted; qty
```

2.2.4.12 getTime

```
currentTime <- getTime
```

getTime keyword.
Gets the ledger time. (You will usually want to immediately bind it to a variable in order to be able to access the value.)
Used to restrict when a choice can be made. For example, with an assert that the time is later than a certain time.

Here's an example of a choice that uses a check on the current time:

```
      Complete : ()

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

2.2.4.13 return

```
return ()
```

return keyword.
Used to return a value from do block that is not of type Update.

Here's an example where two contracts are created in a choice and both their ids are returned as a tuple:

```
do
  firstContract <- create SomeContractTemplate with arg1; arg2
  secondContract <- create SomeContractTemplate with arg1; arg2
  return (firstContract, secondContract)
```

2.2.4.14 let

See the documentation on *Let*.

Let looks similar to binding variables, but it's very different! This code example shows how:

```
do
  -- defines a function, createdContract, taking a single argument that
↪when
  -- called _will_ create the new contract using argument for issuer and
↪owner
  let createContract x = create NameOfContract with issuer = x; owner = x
```

(continues on next page)

(continued from previous page)

```
createContract party1
createContract party2
```

2.2.4.15 this

`this` lets you refer to the current contract from within the choice body. This refers to the contract, *not* the contract ID.

It's useful, for example, if you want to pass the current contract to a helper function outside the template.

2.2.5 Reference: data types

This page gives reference information on DAML's data types:

2.2.5.1 Built-in types

Table of built-in primitive types

Type	For	Example	Notes
`Int`	integers	`1`, `1000000`, `1_000_000`	`Int` values are signed 64-bit integers which represent numbers between `-9,223,372,036,854,775,808` and `9,223,372,036,854,775,807` inclusive. Arithmetic operations raise an error on overflows and division by `0`. To make long numbers more readable you can optionally add underscores.
`Decimal`	short for `Numeric 10`	`1.0`	`Decimal` values are rational numbers with precision 38 and scale 10.
`Numeric n`	fixed point decimal numbers	`1.0`	*Numeric n* values are rational numbers with up to `38` digits. The scale parameter `n` controls the number of digits after the decimal point, so for example, `Numeric 10` values have 10 decimal places, and `Numeric 20` values have 20 decimal places. The value of `n` must be between `0` and `37` inclusive.
`Text`	strings	`"hello"`	`Text` values are strings of characters enclosed by double quotes.
`Bool`	boolean values	`True`, `False`	
`Party`	unicode string representing a party	`alice <- getParty "Alice"`	Every *party* in a DAML system has a unique identifier of type `Party`. To create a value of type `Party`, use binding on the result of calling `getParty`. The party text can only contain alphanumeric characters, -, _ and spaces.
`Date`	models dates	`date 2007 Apr 5`	To create a value of type `Date`, use the function `date` (to get this function, import `DA.Date`).
`Time`	models absolute time (UTC)	`time (date 2007 Apr 5) 14 30 05`	`Time` values have microsecond precision. To create a value of type `Time`, use a `Date` and the function `time` (to get this function, import `DA.Time`).
`RelTime`	models differences between time values	`seconds 1`, `seconds (-2)`	`seconds 1` and `seconds (-2)` represent the values for 1 and -2 seconds. There are no literals for `RelTime`. Instead they are created using one of `days`, `hours`, `minutes` and `seconds` (to get these functions, import `DA.Time`).

Escaping characters

`Text` literals support backslash escapes to include their delimiter (`\"`) and a backslash itself (`\\`).

Time

Definition of time on the ledger is a property of the execution environment. DAML assumes there is a shared understanding of what time is among the stakeholders of contracts.

2.2.5.2 Lists

`[a]` is the built-in data type for a list of elements of type `a`. The empty list is denoted by `[]` and `[1, 3, 2]` is an example of a list of type `[Int]`.

You can also construct lists using `[]` (the empty list) and `::` (which is an operator that appends an element to the front of a list). For example:

```
twoEquivalentListConstructions =
  scenario do
    assert ( [1, 2, 3] == 1 :: 2 :: 3 :: [] )
```

Summing a list

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

2.2.5.3 Records and record types

You declare a new record type using the `data` and `with` keyword:

```
data MyRecord = MyRecord
  with
    label1 : type1
    label2 : type2
    ...
    labelN : typeN
  deriving (Eq, Show)
```

where:

`label1`, `label2`, , `labelN` are *labels*, which must be unique in the record type
`type1`, `type2`, , `typeN` are the types of the fields

There's an alternative way to write record types:

```
data MyRecord = MyRecord { label1 : type1; label2 : type2; ...; labelN :
↪typeN }
  deriving (Eq, Show)
```

The format using `with` and the format using `{ }` are exactly the same syntactically. The main difference is that when you use `with`, you can use newlines and proper indentation to avoid the delimiting semicolons.

The `deriving (Eq, Show)` ensures the data type can be compared (using `==`) and displayed (using `show`). The line starting `deriving` is required for data types used in fields of a `template`.

In general, add the `deriving` unless the data type contains function types (e.g. `Int -> Int`), which cannot be compared or shown.

For example:

```
-- This is a record type with two fields, called first and second,
-- both of type `Int`
data MyRecord = MyRecord with first : Int; second : Int
  deriving (Eq, Show)

-- An example value of this type is:
newRecord = MyRecord with first = 1; second = 2

-- You can also write:
newRecord = MyRecord 1 2
```

Data constructors

You can use `data` keyword to define a new data type, for example `data Floor a = Floor a` for some type `a`.

The first `Floor` in the expression is the *type constructor*. The second `Floor` is a *data constructor* that can be used to specify values of the `Floor Int` type: for example, `Floor 0`, `Floor 1`.

In DAML, data constructors may take *at most one argument*.

An example of a data constructor with zero arguments is `data Empty = Empty {}`. The only value of the `Empty` type is `Empty`.

Note: In `data Confusing = Int`, the `Int` is a data constructor with no arguments. It has nothing to do with the built-in `Int` type.

Accessing record fields

To access the fields of a record type, use dot notation. For example:

```
-- Access the value of the field `first`
val.first

-- Access the value of the field `second`
val.second
```

Updating record fields

You can also use the `with` keyword to create a new record on the basis of an existing replacing select fields.

For example:

```
myRecord = MyRecord with first = 1; second = 2

myRecord2 = myRecord with second = 5
```

produces the new record value `MyRecord with first = 1; second = 5`.

If you have a variable with the same name as the label, DAML lets you use this without assigning it to make things look nicer:

```
-- if you have a variable called `second` equal to 5
second = 5

-- you could construct the same value as before with
myRecord2 = myRecord with second = second

-- or with
myRecord3 = MyRecord with first = 1; second = second

-- but DAML has a nicer way of putting this:
myRecord4 = MyRecord with first = 1; second

-- or even
myRecord5 = r with second
```

Note: The `with` keyword binds more strongly than function application. So for a function, say `return`, either write `return IntegerCoordinate with first = 1; second = 5` or `return (IntegerCoordinate {first = 1; second = 5})`, where the latter expression is enclosed in parentheses.

Parameterized data types

DAML supports parameterized data types.

For example, to express a more general type for 2D coordinates:

```
-- Here, a and b are type parameters.
-- The Coordinate after the data keyword is a type constructor.
data Coordinate a b = Coordinate with first : a; second : b
```

An example of a type that can be constructed with `Coordinate` is `Coordinate Int Int`.

2.2.5.4 Type synonyms

To declare a synonym for a type, use the `type` keyword.

For example:

```
type IntegerTuple = (Int, Int)
```

This makes `IntegerTuple` and `(Int, Int)` synonyms: they have the same type and can be used interchangeably.

You can use the `type` keyword for any type, including *Built-in types*.

Function types

A function's type includes its parameter and result types. A function `foo` with two parameters has type `ParamType1 -> ParamType2 -> ReturnType`.

Note that this can be treated as any other type. You could for instance give it a synonym using `type FooType = ParamType1 -> ParamType2 -> ReturnType`.

2.2.5.5 Algebraic data types

An algebraic data type is a composite type: a type formed by a combination of other types. The enumeration data type is an example. This section introduces more powerful algebraic data types.

Product types

The following data constructor is not valid in DAML: `data AlternativeCoordinate a b = AlternativeCoordinate a b`. This is because data constructors can only have one argument.

To get around this, wrap the values in a *record*: `data Coordinate a b = Coordinate {first: a; second: b}`.

These kinds of types are called *product* types.

A way of thinking about this is that the `Coordinate Int Int` type has a first and second dimension (that is, a 2D product space). By adding an extra type to the record, you get a third dimension, and so on.

Sum types

Sum types capture the notion of being of one kind or another.

An example is the built-in data type `Bool`. This is defined by `data Bool = True | False deriving (Eq,Show)`, where `True` and `False` are data constructors with zero arguments . This means that a `Bool` value is either `True` or `False` and cannot be instantiated with any other value.

Please note that all types which you intend to use as template or choice arguments need to derive at least from *(Eq, Show)*.

A very useful sum type is `data Optional a = None | Some a deriving (Eq,Show)`. It is part of the DAML standard library.

`Optional` captures the concept of a box, which can be empty or contain a value of type `a`.

`Optional` is a sum type constructor taking a type `a` as parameter. It produces the sum type defined by the data constructors `None` and `Some`.

The `Some` data constructor takes one argument, and it expects a value of type `a` as a parameter.

Pattern matching

You can match a value to a specific pattern using the `case` keyword.

The pattern is expressed with data constructors. For example, the `Optional Int` sum type:

```
optionalIntegerToText (x : Optional Int) : Text =
  case x of
    None -> "Box is empty"
    Some val -> "The content of the box is " <> show val

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

In the `optionalIntegerToText` function, the `case` construct first tries to match the `x` argument against the `None` data constructor, and in case of a match, the `"Box is empty"` text is returned. In case of no match, a match is attempted for `x` against the next pattern in the list, i.e., with the `Some` data constructor. In case of a match, the content of the value attached to the `Some` label is bound to the `val` variable, which is then used in the corresponding output text string.

Note that all patterns in the case construct need to be *complete*, i.e., for each `x` there must be at least one pattern that matches. The patterns are tested from top to bottom, and the expression for the first pattern that matches will be executed. Note that _ can be used as a catch-all pattern.

You could also case distinguish a `Bool` variable using the `True` and `False` data constructors and achieve the same behavior as an if-then-else expression.

As an example, the following is an expression for a `Text`:

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

Notice the use of nested pattern matching above.

Note: An underscore was used in place of a variable name. The reason for this is that *DAML Studio* produces a warning for all variables that are not being used. This is useful in detecting unused variables. You can suppress the warning by naming the variable with an initial underscore.

2.2.6 Reference: built-in functions

This page gives reference information on functions for:

2.2.6.1 Working with time

DAML has these built-in functions for working with time:

`datetime`: creates a `Time` given year, month, day, hours, minutes, and seconds as argument.
`subTime`: subtracts one time from another. Returns the `RelTime` difference between `time1` and `time2`.
`addRelTime`: add times. Takes a `Time` and `RelTime` and adds the `RelTime` to the `Time`.
`days`, `hours`, `minutes`, `seconds`: constructs a `RelTime` of the specified length.
`pass`: (in *scenario tests* only) use `pass : RelTime -> Scenario Time` to advance the ledger time by the argument amount. Returns the new time.

2.2.6.2 Working with numbers

DAML has these built-in functions for working with numbers:

`round`: rounds a `Decimal` number to `Int`.
`round d` is the *nearest* `Int` to `d`. Tie-breaks are resolved by rounding away from zero, for example:

```
round 2.5 == 3    round (-2.5) == -3
round 3.4 == 3    round (-3.7) == -4
```

`truncate`: converts a `Decimal` number to `Int`, truncating the value towards zero, for example:

```
truncate 2.2 == 2    truncate (-2.2) == -2
truncate 4.9 == 4    v (-4.9) == -4
```

`intToDecimal`: converts an `Int` to `Decimal`.

The set of numbers expressed by `Decimal` is not closed under division as the result may require more than 10 decimal places to represent. For example, `1.0 / 3.0 == 0.3333...` is a rational number, but not a `Decimal`.

2.2.6.3 Working with text

DAML has these built-in functions for working with text:

`<>` operator: concatenates two `Text` values.
`show` converts a value of the primitive types (`Bool`, `Int`, `Decimal`, `Party`, `Time`, `RelTime`) to a `Text`.

To escape text in DAML strings, use `\`:

Character	How to escape it
`\`	`\\`
`"`	`\"`
`'`	`\'`
Newline	`\n`
Tab	`\t`
Carriage return	`\r`
Unicode (using `!` as an example)	Decimal code: `\33` Octal code: `\o41` Hexadecimal code: `\x21`

2.2.6.4 Working with lists

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

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

Folding

A *fold* takes:

a binary operator
a first *accumulator* value

a list of values

The elements of the list are processed one-by-one (from the left in a `foldl`, or from the right in a `foldr`).

Note: We'd usually recommend using `foldl`, as `foldr` is usually slower. This is because it needs to traverse the whole list before starting to discharge its elements.

Processing goes like this:

1. The binary operator is applied to the first accumulator value and the first element in the list. This produces a second accumulator value.
2. The binary operator is applied to the *second* accumulator value and the second element in the list. This produces a third accumulator value.
3. This continues until there are no more elements in the list. Then, the last accumulator value is returned.

As an example, to sum up a list of integers in DAML:

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

2.2.7 Reference: expressions

This page gives reference information for DAML expressions that are not *updates*:

2.2.7.1 Definitions

Use 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
```

(continues on next page)

(continued from previous page)

```
        with newOwner : Party
        do
          updateOwner newOwner
```

2.2.8 Reference: functions

This page gives reference information on functions in DAML:

DAML is a functional language. It lets you apply functions partially and also have functions that take other functions as arguments. This page discusses these *higher-order functions*.

2.2.8.1 Defining functions

In *Reference: expressions*, the `tubeSurfaceArea` function was defined as:

```
tubeSurfaceArea : Decimal -> Decimal -> Decimal
tubeSurfaceArea r h  =
  2.0 * pi * r * h
```

You can define this function equivalently using lambdas, involving ', a sequence of parameters, and an arrow -> as:

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

2.2.8.2 Partial application

The type of the `tubeSurfaceArea` function described previously, is `Decimal -> Decimal -> Decimal`. An equivalent, but more instructive, way to read its type is: `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 : Decimal) == 3)
    assert (round (3.5 : Decimal) == 4)

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

(continues on next page)

(continued from previous page)

```
    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:

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*:

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 may look different to ones on other DAML Ledgers.

You can use `==` and `/=` on contract identifiers of the same type.

2.2.11 Reference: DAML packages

This page gives reference information on DAML package dependencies:

2.2.11.1 Building DAML archives

When a DAML project is compiled, the compiler produces a *DAML archive*. These are platform-independent packages of compiled DAML code that can be uploaded to a DAML ledger or imported in other DAML projects.

DAML archives have a `.dar` file ending. By default, when you run `daml build`, it will generate the `.dar` file in the `.daml/dist` folder in the project root folder. For example, running `daml build` in project `foo` with project version `0.0.1` will result in a DAML archive `.daml/dist/foo-0.0.1.dar`.

You can specify a different path for the DAML archive by using the `-o` flag:

```
daml build -o foo.dar
```

For details on how to upload a DAML archive to the ledger, see the *deploy documentation*. The rest of this page will focus on how to import a DAML packages in other DAML projects.

2.2.11.2 Inspecting DARs

To inspect a DAR and get information about the packages inside it, you can use the `daml damlc inspect-dar` command. This is often useful to find the package id of the project you just built.

You can run `daml damlc inspect-dar /path/to/your.dar` to get a human-readable listing of the files inside it and a list of packages and their package ids. Here is a (shortened) example output:

```
$ daml damlc inspect-dar .daml/dist/create-daml-app-0.1.0.dar
DAR archive contains the following files:

create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/create-
↪daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d.dalf
create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/daml-
↪prim-75b070729b1fbd37a618493652121b0d6f5983b787e35179e52d048db70e9f15.
↪dalf
create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/daml-
↪stdlib-0.0.0-
↪a535cbc3657b8df953a50aaef5a4cd224574549c83ca4377e8219aadea14f21a.dalf
create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/daml-
↪stdlib-DA-Internal-Template-
↪d14e08374fc7197d6a0de468c968ae8ba3aadbf9315476fd39071831f5923662.dalf
create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/data/
↪create-daml-app-0.1.0.conf
create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/User.
↪daml
create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/User.hi
create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/User.hie
```

(continues on next page)

(continued from previous page)

```
META-INF/MANIFEST.MF

DAR archive contains the following packages:

create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d
↪"29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d"
daml-stdlib-DA-Internal-Template-
↪d14e08374fc7197d6a0de468c968ae8ba3aadbf9315476fd39071831f5923662
↪"d14e08374fc7197d6a0de468c968ae8ba3aadbf9315476fd39071831f5923662"
daml-prim-75b070729b1fbd37a618493652121b0d6f5983b787e35179e52d048db70e9f15
↪"75b070729b1fbd37a618493652121b0d6f5983b787e35179e52d048db70e9f15"
daml-stdlib-0.0.0-
↪a535cbc3657b8df953a50aaef5a4cd224574549c83ca4377e8219aadea14f21a
↪"a535cbc3657b8df953a50aaef5a4cd224574549c83ca4377e8219aadea14f21a"
```

In addition to the human-readable output, you can also get the output as JSON. This is easier to consume programatically and it is more robust to changes across SDK versions:

```
$ daml damlc inspect-dar --json .daml/dist/create-daml-app-0.1.0.dar
{
    "packages": {
        "29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d
↪": {
            "path": "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/create-
↪daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d.dalf",
            "name": "create-daml-app",
            "version": "0.1.0"
        },
        "d14e08374fc7197d6a0de468c968ae8ba3aadbf9315476fd39071831f5923662
↪": {
            "path": "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/daml-
↪stdlib-DA-Internal-Template-
↪d14e08374fc7197d6a0de468c968ae8ba3aadbf9315476fd39071831f5923662.dalf",
            "name": null,
            "version": null
        },
        "75b070729b1fbd37a618493652121b0d6f5983b787e35179e52d048db70e9f15
↪": {
            "path": "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/daml-
↪prim-75b070729b1fbd37a618493652121b0d6f5983b787e35179e52d048db70e9f15.
↪dalf",
            "name": "daml-prim",
            "version": "0.0.0"
        },
```

(continues on next page)

(continued from previous page)

```
        "a535cbc3657b8df953a50aaef5a4cd224574549c83ca4377e8219aadea14f21a
↪": {
            "path": "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/daml-
↪stdlib-0.0.0-
↪a535cbc3657b8df953a50aaef5a4cd224574549c83ca4377e8219aadea14f21a.dalf",
            "name": "daml-stdlib",
            "version": "0.0.0"
        }
    },
    "main_package_id":
↪"29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d",
    "files": [
        "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/create-
↪daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d.dalf",
        "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/daml-
↪prim-75b070729b1fbd37a618493652121b0d6f5983b787e35179e52d048db70e9f15.
↪dalf",
        "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/daml-
↪stdlib-0.0.0-
↪a535cbc3657b8df953a50aaef5a4cd224574549c83ca4377e8219aadea14f21a.dalf",
        "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/daml-
↪stdlib-DA-Internal-Template-
↪d14e08374fc7197d6a0de468c968ae8ba3aadbf9315476fd39071831f5923662.dalf",
        "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/data/
↪create-daml-app-0.1.0.conf",
        "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/User.
↪daml",
        "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/User.hi
↪",
        "create-daml-app-0.1.0-
↪29b501bcf541a40e9f75750246874e0a35de72e00616372da435e4b69966db5d/User.hie
↪",
        "META-INF/MANIFEST.MF"
    ]
}
```

Note that `name` and `version` will be `null` for packages in DAML-LF < 1.8.

2.2.11.3 Importing DAML packages

There are two ways to import a DAML package in a project: via `dependencies`, and via `data-dependencies`. They each have certain advantages and disadvantages. To summarize:

> `dependencies` allow you to import a DAML archive as a library. The definitions in the dependency will all be made available to the importing project. However, the dependency must be compiled with the same DAML SDK version, so this method is only suitable for breaking up large projects into smaller projects that depend on each other, or to reuse existing libraries.
> `data-dependencies` allow you to import a DAML archive (.dar) or a DAML-LF package (.dalf), including packages that have already been deployed to a ledger. These packages can be compiled with any previous SDK version. On the other hand, not all definitions can be carried over perfectly, since the DAML interface needs to be reconstructed from the binary.

The following sections will cover these two approaches in more depth.

Importing a DAML package via dependencies

A DAML project can declare a DAML archive as a dependency in the `dependencies` field of `daml.yaml`. This lets you import modules and reuse definitions from another DAML project. The main limitation of this method is that the dependency must be for the same SDK version as the importing project.

Let's go through an example. Suppose you have an existing DAML project `foo`, located at `/home/user/foo`, and you want to use it as a dependency in a project `bar`, located at `/home/user/bar`.

To do so, you first need to generate the DAML archive of `foo`. Go into `/home/user/foo` and run `daml build -o foo.dar`. This will create the DAML archive, `/home/user/foo/foo.dar`.

Next, we will update the project config for `bar` to use the generated DAML archive as a depndency. Go into `/home/user/bar` and change the `dependencies` field in `daml.yaml` to point to the created *DAML archive*:

```
dependencies:
  - daml-prim
  - daml-stdlib
  - ../foo/foo.dar
```

The import path can also be absolute, for example, by changing the last line to:

```
- /home/user/foo/foo.dar
```

When you run `daml build` in `bar` project, the compiler will make the definitions in `foo.dar` available for importing. For example, if `foo` exports the module `Foo`, you can import it in the usual way:

```
import Foo
```

By default, all modules of `foo` are made available when importing `foo` as a dependency. To limit which modules of `foo` get exported, you may add an `exposed-modules` field in the `daml.yaml` file for `foo`:

```
exposed-modules:
- Foo
```

Importing a DAML archive via data-dependencies

You can import a DAML archive (.dar) or DAML-LF package (.dalf) using `data-dependencies`. Unlike `dependencies`, this can be used when the DAML SDK versions do not match.

For example, you can import `foo.dar` as follows:

```yaml
dependencies:
- daml-prim
- daml-stdlib
data-dependencies:
- ../foo/foo.dar
```

When importing packages this way, the DAML compiler will try to reconstruct the original DAML interface from the compiled binaries. However, to allow `data-dependencies` to work across SDK versions, the compiler has to abstract over some details which are not compatible across SDK versions. This means that there are some DAML features that cannot be recovered when using `data-dependencies`. In particular:

1. Export lists cannot be recovered, so imports via `data-dependencies` can access definitions that were originally hidden. This means it is up to the importing module to respect the data abstraction of the original module. Note that this is the same for all code that runs on the ledger, since the ledger does not provide special support for data abstraction.
2. If you have a `dependency` that limits the modules that can be accessed via `exposed-modules`, you can get an error if you also have a `data-dependency` that references something from the hidden modules (even if it is only reexported). Since `exposed-modules` are not available on the ledger in general, we recommend to not make use of them and instead rely on naming conventions (e.g., suffix module names with `.Internal`) to make it clear which modules are part of the public API.
3. Prior to DAML-LF version 1.8, typeclasses could not be reconstructed. This means if you have a package that is compiled with an older version of DAML-LF, typeclasses and typeclass instances will not be carried over via data-dependencies, and you won't be able to call functions that rely on typeclass instances. This includes the template functions, such as `create`, `signatory`, and `exercise`, as these rely on typeclass instances.
4. Starting from DAML-LF version 1.8, when possible, typeclass instances will be reconstructed by re-using the typeclass definitions from dependencies, such as the typeclasses exported in `daml-stdlib`. However, if the typeclass signature has changed, you will get an instance for a reconstructed typeclass instead, which will not interoperate with code from dependencies. Furthermore, if the typeclass definition uses the `FunctionalDependencies` language extension, this may cause additional problems, since the functional dependencies cannot be recovered. So this is something to keep in mind when redefining typeclasses and when using `FunctionalDependencies`.
5. Certain advanced type system features cannot be reconstructed. In particular, `DA.Generics` and `DeriveGeneric` cannot be reconstructed. This may result in certain definitions being unavailable when importing a module that uses these advanced features.

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

2.2.11.4 Handling module name collisions

Sometimes you will have multiple packages with the same module name. In that case, a simple import will fail, since the compiler doesn't know which version of the module to load. Fortunately, there are a few tools you can use to approach this problem.

The first is to use package qualified imports. Supposing you have packages with different names, foo and bar, which both expose a module X. You can select which one you want with a package qualified import.

To get X from foo:

```
import "foo" X
```

To get X from bar:

```
import "bar" X
```

To get both, you need to rename the module as you perform the import:

```
import "foo" X as FooX
import "bar" X as BarX
```

Sometimes, package qualified imports will not help, because you are importing two packages with the same name. For example, if you're loading different versions of the same package. To handle this case, you need the --package build option.

Suppose you are importing packages foo-1.0.0 and foo-2.0.0. Notice they have the same name foo but different versions. To get modules that are exposed in both packages, you will need to provide module aliases. You can do this by passing the --package build option. Open daml.yaml and add the following build-options:

```
build-options:
- '--package'
- 'foo-1.0.0 with (X as Foo1.X)'
- '--package'
- 'foo-2.0.0 with (X as Foo2.X)'
```

This will alias the X in foo-1.0.0 as Foo1.X, and alias the X in foo-2.0.0 as Foo2.X. Now you will be able to import both X by using the new names:

```
import qualified Foo1.X
import qualified Foo2.X
```

It is also possible to add a prefix to all modules in a package using the module-prefixes field in your daml.yaml. This is partiuclarly useful for upgrades where you can map all modules of version v of your package under V$v. For the example above you can use the following:

```
module-prefixes:
  foo-1.0.0: Foo1
  foo-2.0.0: Foo2
```

That will allow you to import module X from package foo-1.0.0 as Foo1.X and X from package -foo-2.0.0 as Foo2.

You can also use more complex module prefixes, e.g., foo-1.0.0: Foo1.Bar which will make module X available under Foo1.Bar.X.

2.2.12 Contract keys

Contract keys are an optional addition to templates. They let you specify a way of uniquely identifying 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 serializable expression that does **not** contain contract IDs. However, it **must** include every party that you want to use as a `maintainer` (see *Specifying maintainers* below).

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

2.2.12.2 Specifying maintainers

If you specify a contract key for a template, you must also specify a `maintainer` or maintainers, in a similar way to specifying signatories or observers. The maintainers own the key in the same way the signatories own a contract. Just like signatories of contracts prevent double spends or use of false contract data, maintainers of keys prevent double allocation or incorrect lookups. Since the key is part of the contract, the maintainers **must** be signatories of the contract. However, maintainers are computed from the `key` instead of the template arguments. In the example above, the `bank` is ultimately the maintainer of the key.

Uniqueness of keys is guaranteed per template. Since multiple templates may use the same key type, some key-related functions must be annotated using the `@ContractType` as shown in the examples below.

When you are writing DAML models, the maintainers matter since they affect authorization – much like signatories and observers. You don't need to do anything to maintain the keys. In the above example, it is guaranteed that there can only be one `Account` with a given `number` at a given `bank`.

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

The primary purpose of contract keys is to provide a stable, and possibly meaningful, identifier that can be used in DAML to fetch contracts. There are two functions to perform such lookups: *fetchByKey* and *lookupByKey*. Both types of lookup are performed at interpretation time on the submitting Participant Node, on a best-effort basis. Currently, that best-effort means lookups only return contracts if the submitting Party is a stakeholder of that contract.

In particular, the above means that if multiple commands are submitted simultaneously, all using contract lookups to find and consume a given contract, there will be contention between these commands, and at most one will succeed.

Limiting key usage to stakeholders also means that keys cannot be used to access a divulged contract, i.e. there can be cases where *fetch* succeeds and *fetchByKey* does not. See the example at the end of this section for details.

fetchByKey

```
(fetchedContractId, fetchedContract) <- fetchByKey @ContractType
contractKey
```

Use `fetchByKey` to fetch the ID and data of the contract with the specified key. It is an alternative to `fetch` and behaves the same in most ways.

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

Like `fetch`, `fetchByKey` needs to be authorized by at least one stakeholder.

`fetchByKey` fails and aborts the transaction if:

> The submitting Party is not a stakeholder on a contract with the given key, or
> A contract was found, but the `fetchByKey` violates the authorization rule, meaning no stakeholder authorized the `fetch`..

This means that if it fails, it doesn't guarantee that a contract with that key doesn't exist, just that the submitting Party doesn't know about it, or there are issues with authorization.

visibleByKey

```
boolean <- visibleByKey @ContractType contractKey
```

Use `visibleByKey` to check whether you can see an active contract for the given key with the current authorizations. If the contract exists and you have permission to see it, returns `True`, otherwise returns `False`.

To clarify, ignoring contention:

1. `visibleByKey` will return `True` if all of these are true: there exists a contract for the given key, the submitter is a stakeholder on that contract, and at the point of call we have the authorization of **all** of the maintainers of the key.
2. `visibleByKey` will return `False` if all of those are true: there is no contract for the given key, and at the point of call we have authorization from **all** the maintainers of the key.
3. `visibleByKey` will abort the transaction at interpretation time if, at the point of call, we are missing the authorization from any one maintainer of the key.
4. `visibleByKey` will fail at validation time (after returning `False` at interpretation time) if all of these are true: at the point of call, we have the authorization of **all** the maintainers, and a valid contract exists for the given key, but the submitter is not a stakeholder on that contract.

While it may at first seem too restrictive to require **all** maintainers to authorize the call, this is actually required in order to validate negative lookups. In the positive case, when you can see the contract, it's easy for the transaction to mention which contract it found, and therefore for validators to check that this contract does indeed exist, and is active as of the time of executing the transaction.

For the negative case, however, the transaction submitted for execution cannot say _which_ contract it has not found (as, by definition, it has not found it, and it may not even exist). Still, validators have to be able to reproduce the result of not finding the contract, and therefore they need to be able to look for it, which means having the authorization to ask the maintainers about it.

lookupByKey

`optionalContractId <- lookupByKey @ContractType contractKey`

Use `lookupByKey` to check whether a contract with the specified key exists. If it does exist, `lookupByKey` returns the `Some contractId`, where `contractId` is the ID of the contract; otherwise, it returns `None`.

`lookupByKey` is conceptually equivalent to

```
lookupByKey : forall c k. (HasFetchByKey c k) => k -> Update (Optional (ContractId c))
lookupByKey k = do
  visible <- visibleByKey @c k
  if visible then do
    (contractId, _ignoredContract) <- fetchByKey @c k
    return $ Some contractId
  else
    return None
```

Therefore, `lookupByKey` needs all the same authorizations as *visibleByKey*, for the same reasons, and fails in the same cases.

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

2.2.12.4 exerciseByKey

`exerciseByKey @ContractType contractKey`

Use `exerciseByKey` to exercise a choice on a contract identified by its `key` (compared to `exercise`, which lets you exercise a contract identified by its `ContractId`). To run `exerciseByKey` you need authorization from the controllers of the choice and at least one stakeholder. This is equivalent to the authorization needed to do a `fetchByKey` followed by an `exercise`.

2.2.12.5 Example

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

```
-- Copyright (c) 2020 Digital Asset (Switzerland) GmbH and/or its affiliates. All rights reserved.
-- SPDX-License-Identifier: Apache-2.0

module Keys where
```

(continues on next page)

(continued from previous page)

```
import DA.Optional

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

    key sig : Party
    maintainer key

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

    controller sig can
      nonconsuming DivulgeKeyed
        : Keyed
        with
          keyedCid : ContractId Keyed
        do
          fetch keyedCid

template Delegation
  with
    sig : Party
    delegees : [Party]
  where
    signatory sig
    observer delegees

    nonconsuming choice CreateKeyed
      : ContractId Keyed
      with
        delegee : Party
        obs : Party
      controller delegee
      do
        create Keyed with sig; obs

    nonconsuming choice ArchiveKeyed
      : ()
      with
```

(continues on next page)

(continued from previous page)

```
        delegee : Party
        keyedCid : ContractId Keyed
      controller delegee
      do
        archive keyedCid

    nonconsuming choice UnkeyedFetch
      : Keyed
      with
        cid : ContractId Keyed
        delegee : Party
      controller delegee
      do
        fetch cid

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

    nonconsuming choice LookupKeyed
      : Optional (ContractId Keyed)
      with
        lookupKey : Party
        delegee : Party
      controller delegee
      do
        lookupByKey @Keyed lookupKey

    nonconsuming choice FetchKeyed
      : (ContractId Keyed, Keyed)
      with
        lookupKey : Party
        delegee : Party
      controller delegee
      do
        fetchByKey @Keyed lookupKey

lookupTest = scenario do

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

(continues on next page)

(continued from previous page)

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

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

-- TESTING LOOKUPS AND FETCHES

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

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

(continues on next page)

(continued from previous page)

```
    assert (mcid == Some keyedCid)

  -- Divulgee _can_ fetch the contract directly
  submit divulgee do
    exercise obsDelegationCid UnkeyedFetch with
        delegee = divulgee
        cid = keyedCid
  -- Divulgee can't fetch through the key
  submitMustFail divulgee do fetchByKey @Keyed sig
  -- Divulgee can't see
  submitMustFail divulgee do visibleByKey @Keyed sig
  -- Divulgee can't see with stakeholder authority
  submitMustFail divulgee do
    exercise obsDelegationCid VisibleKeyed with
        delegee = divulgee
        key = sig
  -- Divulgee can't lookup
  submitMustFail divulgee do lookupByKey @Keyed sig
  -- Divulgee can't lookup with stakeholder authority
  submitMustFail divulgee do
    exercise obsDelegationCid LookupKeyed with
        delegee = divulgee
        lookupKey = sig
  -- Divulgee can't do positive lookup with maintainer authority.
  submitMustFail divulgee do
    b <- exercise sigDelegationCid VisibleKeyed with
      delegee = divulgee
      key = sig
    assert $ not b
  -- Divulgee can't do positive lookup with maintainer authority.
  -- Note that the lookup returns `None` so the assertion passes.
  -- If the assertion is changed to `isSome`, the assertion fails,
  -- which means the error message changes. The reason is that the
  -- assertion is checked at interpretation time, before the lookup
  -- is checked at validation time.
  submitMustFail divulgee do
    mcid <- exercise sigDelegationCid LookupKeyed with
      delegee = divulgee
      lookupKey = sig
    assert (isNone mcid)
  -- Divulgee can't fetch with stakeholder authority
  submitMustFail divulgee do
    (cid, keyed) <- exercise obsDelegationCid FetchKeyed with
      delegee = divulgee
      lookupKey = sig
    assert (keyedCid == cid)

  -- Blind party can't fetch
  submitMustFail blind do fetchByKey @Keyed sig
```

(continues on next page)

(continued from previous page)

```
  -- Blind party can't see
  submitMustFail blind do visibleByKey @Keyed sig
  -- Blind party can't see with stakeholder authority
  submitMustFail blind do
    exercise obsDelegationCid VisibleKeyed with
      delegee = blind
      key = sig
  -- Blind party can't see with maintainer authority
  submitMustFail blind do
    b <- exercise sigDelegationCid VisibleKeyed with
      delegee = blind
      key = sig
    assert $ not b
  -- Blind party can't lookup
  submitMustFail blind do lookupByKey @Keyed sig
  -- Blind party can't lookup with stakeholder authority
  submitMustFail blind do
    exercise obsDelegationCid LookupKeyed with
      delegee = blind
      lookupKey = sig
  -- Blind party can't lookup with maintainer authority.
  -- The lookup initially returns `None`, but is rejected at
  -- validation time
  submitMustFail blind do
    mcid <- exercise sigDelegationCid LookupKeyed with
      delegee = blind
      lookupKey = sig
    assert (isNone mcid)
  -- Blind party can't fetch with stakeholder authority as lookup is negative
  submitMustFail blind do
    exercise obsDelegationCid FetchKeyed with
      delegee = blind
      lookupKey = sig
  -- Blind party can see nonexistence of a contract
  submit blind do
    b <- exercise obsDelegationCid VisibleKeyed with
      delegee = blind
      key = obs
    assert $ not b
  -- Blind can do a negative lookup on a truly nonexistant contract
  submit blind do
    mcid <- exercise obsDelegationCid LookupKeyed with
      delegee = blind
      lookupKey = obs
    assert (isNone mcid)
  -- TESTING CREATES AND ARCHIVES

  -- Divulgee can archive
```

(continues on next page)

(continued from previous page)

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

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

return ()
```

2.3 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:

2.3.1 Scenario syntax

2.3.1.1 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.

2.3.1.2 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.

2.3.1.3 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.

2.3.1.4 Full syntax

For detailed syntax, see *Reference: scenarios*.

2.3.2 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.

2.3.3 Examples

2.3.3.1 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
```

(continues on next page)

(continued from previous page)

```
        -- 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).

2.3.3.2 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.

2.3.3.3 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
      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.

2.4 Troubleshooting

2.4.1 Error: "<X> is not authorized to commit an update"

This error occurs when there are multiple obligables on a contract.

A cornerstone of DAML is that you cannot create a contract that will force some other party (or parties) into an obligation. This error means that a party is trying to do something that would force another parties into an agreement without their consent.

To solve this, make sure each party is entering into the contract freely by exercising a choice. A good way of ensuring this is the initial and accept pattern: see the DAML patterns for more details.

2.4.2 Error "Argument is not of serializable type"

This error occurs when you're using a function as a parameter to a template. For example, here is a contract that creates a `Payout` controller by a receiver's supervisor:

```
template SupervisedPayout
  with
    supervisor : Party -> Party
    receiver   : Party
    giver      : Party
    amount     : Decimal
  where
    controller (supervisor receiver) can
      SupervisedPayout_Call
        returning ContractId Payout
        to create Payout with giver; receiver; amount
```

Hovering over the compilation error displays:

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

2.4.3 Modelling questions

2.4.3.1 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.

2.4.3.2 How to model rights

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

2.4.3.3 How to void a contract

To allow voiding a contract, provide a choice that does not create any new contracts. DAML contracts are archived (but not deleted) when a consuming choice is made - so exercising the choice effectively voids the contract.

However, you should bear in mind who is allowed to void a contract, especially without the re-sought consent of the other signatories.

2.4.3.4 How to represent off-ledger parties

You'd need to do this if you can't set up all parties as ledger participants, because the DAML `Party` type gets associated with a cryptographic key and can so only be used with parties that have been set up accordingly.

To model off-ledger parties in DAML, they must be represented on-ledger by a participant who can sign on their behalf. You could represent them with an ordinary `Text` argument.

This isn't very private, so you could use a numeric ID/an accountId to identify the off-ledger client.

2.4.3.5 How to limit a choice by time

Some rights have a time limit: either a time by which it must be exercised or a time before which it cannot be exercised.

You can use `getTime` to get the current time, and compare your desired time to it. Use `assert` to abort the choice if your time condition is not met.

2.4.3.6 How to model a mandatory action

If you want to ensure that a party takes some action within a given time period. Might want to incur a penalty if they don't - because that would breach the contract.

For example: an Invoice that must be paid by a certain date, with a penalty (could be something like an added interest charge or a penalty fee). To do this, you could have a time-limited Penalty choice that can only be exercised *after* the time period has expired.

However, note that the penalty action can only ever create another contract on the ledger, which represents an agreement between all parties that the initial contract has been breached. Ultimately, the recourse for any breach is legal action of some kind. What DAML provides is provable violation of the agreement.

2.4.3.7 When to use Optional

The `Optional` type, from the standard library, to indicate that a value is optional, i.e, that in some cases it may be missing.

In functional languages, `Optional` is a better way of indicating a missing value than using the more familiar value NULL , present in imperative languages like Java.

To use `Optional`, include `Optional.daml` from the standard library:

```
import DA.Optional
```

Then, you can create `Optional` values like this:

```
Some "Some text"    -- Optional value exists.
None                -- Optional value does not exist.
```

You can test for existence in various ways:

```
-- isSome returns True if there is a value.
if isSome m
  then "Yes"
  else "No"
-- The inverse is isNone.
if isNone m
  then "No"
  else "Yes"
```

If you need to extract the value, use the `optional` function.

It returns a value of a defined type, and takes a `Optional` value and a function that can transform the value contained in a `Some` value of the `Optional` to that type. If it is missing `optional` also takes a value of the return type (the default value), which will be returned if the `Optional` value is `None`

```
let f = \ (i : Int) -> "The number is " <> (show i)
let t = optional "No number" f someValue
```

If `optionalValue` is `Some 5`, the value of `t` would be `"The number is 5"`. If it was `None`, `t` would be `"No number"`. Note that with `optional`, it is possible to return a different type from that contained in the `Optional` value. This makes the `Optional` type very flexible.

There are many other functions in Optional.daml that let you perform familiar functional operations on structures that contain `Optional` values - such as `map`, `filter`, etc. on `Lists` of `Optional` values.

2.4.4 Testing questions

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

Use a `submit` block and a `fetch` operation. The `submit` block tests that the contract (as a `ContractId`) is visible to that party, and the `fetch` tests that it is valid, i.e., that the contract does exist.

For example, if we wanted to test for the existence and visibility of an `Invoice`, visible to 'Alice', whose ContractId is bound to *invoiceCid*, we could say:

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

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

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

using an equality test and an `assert`:

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

2.4.4.2 How to test that an update action cannot be committed

Use the `submitMustFail` function. This is similar in form to the `submit` function, but is an assertion that an update will fail if attempted by some Party.

2.5 Good design patterns

Patterns have been useful in the programming world, as both a source of design inspiration, and a document of good design practices. This document is a catalog of DAML patterns intended to provide the same facility in the DA/DAML application world.

You can checkout the examples locally via `daml new daml-patterns --template daml-patterns`.

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

Multiple party agreement The Multiple Party Agreement pattern uses a Pending contract as a wrapper for the Agreement contract. Any one of the signatory parties can kick off the workflow by creating a Pending contract on the ledger, filling in themselves in all the signatory fields. The Agreement contract is not created on the ledger until all parties have agreed to the Pending contract, and replaced the initiator's signature with their own.

Delegation The Delegation pattern gives one party the right to exercise a choice on behalf of another party. The agent can control a contract 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.

2.5.1 Initiate and Accept

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

2.5.1.1 Motivation

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

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

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

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

2.5.1.2 Implementation

The Initiate and Accept pattern in general involves 2 contracts:

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

```
template CoinMaster
  with
    issuer: Party
  where
    signatory issuer

    controller issuer can
      nonconsuming Invite : ContractId CoinIssueProposal
        with owner: Party
        do create CoinIssueProposal
             with coinAgreement = CoinIssueAgreement with issuer; owner
```

The *CoinIssueProposal* contract has *Issuer* as the signatory, and *Owner* as the controller to the *Accept* choice. In its complete form, the *CoinIssueProposal* contract should define all choices available to the owner, i.e. Accept, Reject or Counter (e.g. re-negotiate terms).

```
template CoinIssueProposal
  with
    coinAgreement: CoinIssueAgreement
  where
    signatory coinAgreement.issuer

    controller coinAgreement.owner can
      AcceptCoinProposal
        : ContractId CoinIssueAgreement
        do create coinAgreement
```

Result contract Once the owner exercises the *AcceptCoinProposal* choice on the initiate contract to express their consent, it returns a result contract representing the agreement between the two parties. In this example, the result contract is of type *CoinIssueAgreement*. Note, it has both *issuer* and *owner* as the signatories, implying they both need to consent to the creation of this contract. Both parties could be controller(s) on the result contract, depending on the business case.

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

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

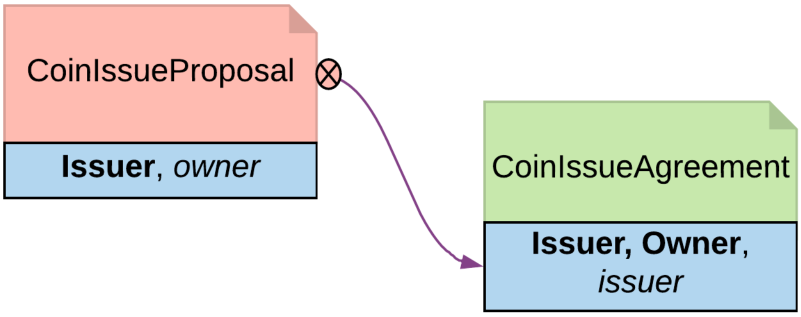


Fig. 1: Initiate and Accept pattern diagram

2.5.1.3 Trade-offs

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

2.5.2 Multiple party agreement

The Multiple Party Agreement pattern uses a Pending contract as a wrapper for the Agreement contract. Any one of the signatory parties can kick off the workflow by creating a Pending contract on the ledger, filling in themselves in all the signatory fields. The Agreement contract is not created on the ledger until all parties have agreed to the Pending contract, and replaced the initiator's signature with their own.

2.5.2.1 Motivation

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

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

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

2.5.2.2 Implementation

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

```
template Agreement
  with
    signatories: [Party]
```

(continues on next page)

(continued from previous page)

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

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

The *Pending* contract has a list of parties who have signed it, and a list of parties who have yet to sign it. If you add these lists together, it has to be the same set of parties as the `signatories` of the *Agreement* contract.

All of the `toSign` parties have the choice to `Sign`. This choice checks that the party is indeed a member of `toSign`, then creates a new instance of the *Pending* contract where they have been moved to the `signed` list.

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
          assert (sort alreadySigned == sort finalContract.signatories)
          create finalContract
```

Collecting the signatures in practice Since the final Pending contract has multiple signatories, **it cannot be created in that state by any one stakeholder**.

However, a party can create a pending contract, with all of the other parties in the `toSign` list.

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

  -- Parties cannot create a contract already signed by someone else
  initialFailTest <- person1 `submitMustFail` do
    create Pending with finalContract; alreadySigned = [person1, person2]

  -- Any party can create a Pending contract provided they list themselves as the only signatory
  pending <- person1 `submit` do
    create Pending with finalContract; alreadySigned = [person1]
```

Once the Pending contract is created, the other parties can sign it. For simplicity, the example code only has choices to express consensus (but you might want to add choices to Accept, Reject, or Negotiate).

```
  -- Each signatory of the finalContract can Sign the Pending contract
  pending <- person2 `submit` do
    exercise pending Sign with signer = person2
  pending <- person3 `submit` do
    exercise pending Sign with signer = person3
  pending <- person4 `submit` do
    exercise pending Sign with signer = person4

  -- A party can't sign the Pending contract twice
  pendingFailTest <- person3 `submitMustFail` do
    exercise pending Sign with signer = person3
  -- A party can't sign on behalf of someone else
  pendingFailTest <- person3 `submitMustFail` do
    exercise pending Sign with signer = person4
```

Once all of the parties have signed the Pending contract, any of them can then exercise the `Finalize` choice. This creates the Agreement contract on the ledger.

```
  person1 `submit` do
    exercise pending Finalize with signer = person1
```

2.5.3 Delegation

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

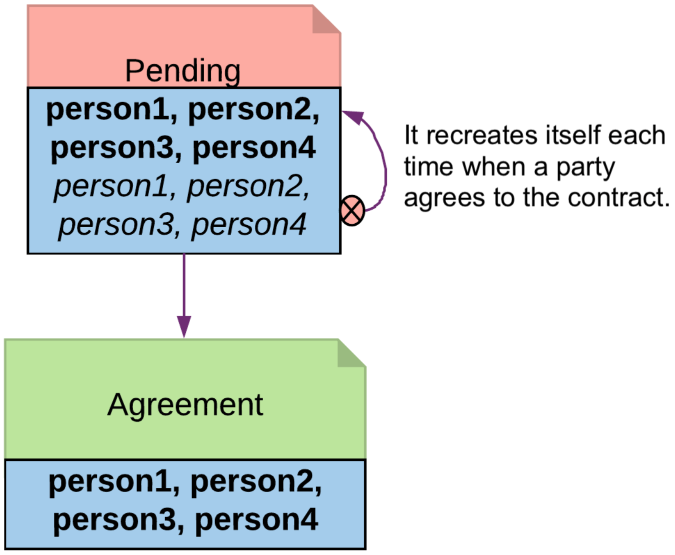

Pending
person1, person2, person3, person4
person1, person2, person3, person4
It recreates itself each time when a party agrees to the contract.
Agreement
person1, person2, person3, person4

Fig. 2: Multiple Party Agreement Diagram

2.5.3.1 Motivation

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

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

2.5.3.2 Implementation

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

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

    controller owner can
```

(continues on next page)

(continued from previous page)

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

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

   Disclose : ContractId Coin
    with p : Party
    do create this with delegates = p :: delegates

  --a coin can only be archived by the issuer under the condition that the issuer is the owner of the coin. This ensures the issuer cannot archive coins at will.
  controller issuer can
   Archives
    : ()
    do assert (issuer == owner)
```

Delegation Contract

Principal, the original coin owner, is the signatory of delegation contract *CoinPoA*. This signatory is required to authorize the *Transfer* choice on *coin*.

```
template CoinPoA
 with
  attorney: Party
  principal: Party
 where
  signatory principal

  controller principal can
   WithdrawPoA
    : ()
    do return ()
```

Whether or not the *Attorney* party should be a signatory of *CoinPoA* is subject to the business agreements between *Principal* and *Attorney*. For simplicity, in this example, *Attorney* is not a signatory.

Attorney is the controller of the Delegation choice on the contract. Within the choice, *Principal* exercises the choice *Transfer* on the Coin contract.

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

(continues on next page)

(continued from previous page)

```
        do
          exercise coinId Transfer with newOwner
```

Coin contracts need to be disclosed to *Attorney* before they can be used in an exercise of *Transfer*. This can be done by adding *Attorney* to *Coin* as an Observer. This can be done dynamically, for any specific *Coin*, by making the observers a *List*, and adding a choice to add a party to that List:

```
      Disclose : ContractId Coin
        with p : Party
        do create this with delegates = p :: delegates
```

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

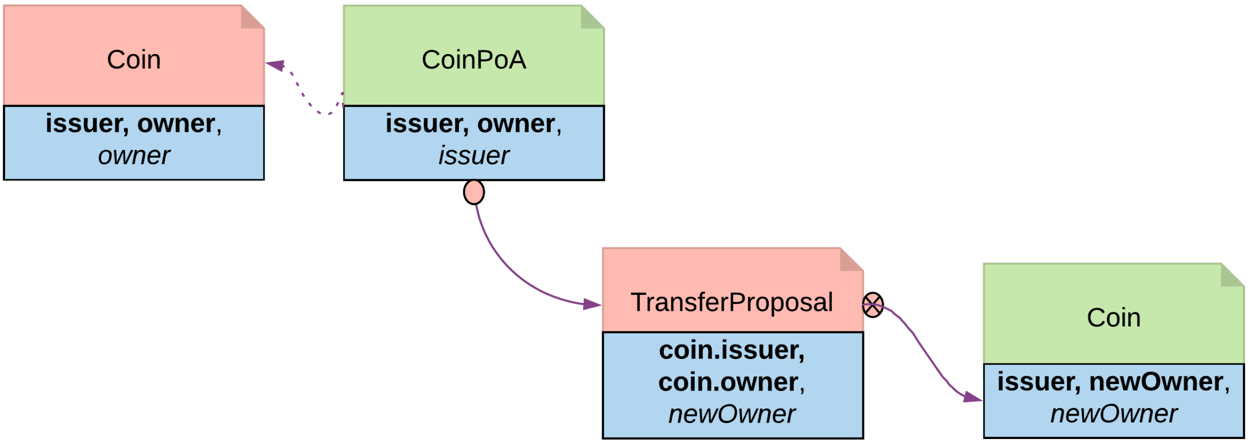


Fig. 3: Delegation pattern diagram

2.5.4 Authorization

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

2.5.4.1 Motivation

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

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

2.5.4.2 Authorization

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

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

    controller owner can

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

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

      Disclose : ContractId Coin
        with p : Party
        do create this with delegates = p :: delegates

    --a coin can only be archived by the issuer under the condition that the issuer is the owner of the coin. This ensures the issuer cannot archive coins at will.
    controller issuer can
      Archives
        : ()
        do assert (issuer == owner)
```

This is may be insufficient since the issuer has no means to ensure the newOwner is an accredited company. The following changes fix this deficiency.

Authorization contract The below shows an authorization contract *CoinOwnerAuthorization*. In this example, the issuer is the only signatory so it can be easily created on the ledger. Owner is an observer on the contract to ensure they can see and use the authorization.

```
template CoinOwnerAuthorization
  with
    owner: Party
    issuer: Party
  where
    signatory issuer
    observer owner
```

(continues on next page)

(continued from previous page)

```
    controller issuer can
      WithdrawAuthorization
        : ()
        do return ()
```

Authorization contracts can have much more advanced business logic, but in its simplest form, *CoinOwnerAuthorization* serves its main purpose, which is to prove the owner is a warranted coin owner.

TransferProposal contract In the TransferProposal contract, the Accept choice checks that newOwner has proper authorization. A *CoinOwnerAuthorization* for the new owner has to be supplied and is checked by the two assert statements in the choice before a coin can be transferred.

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

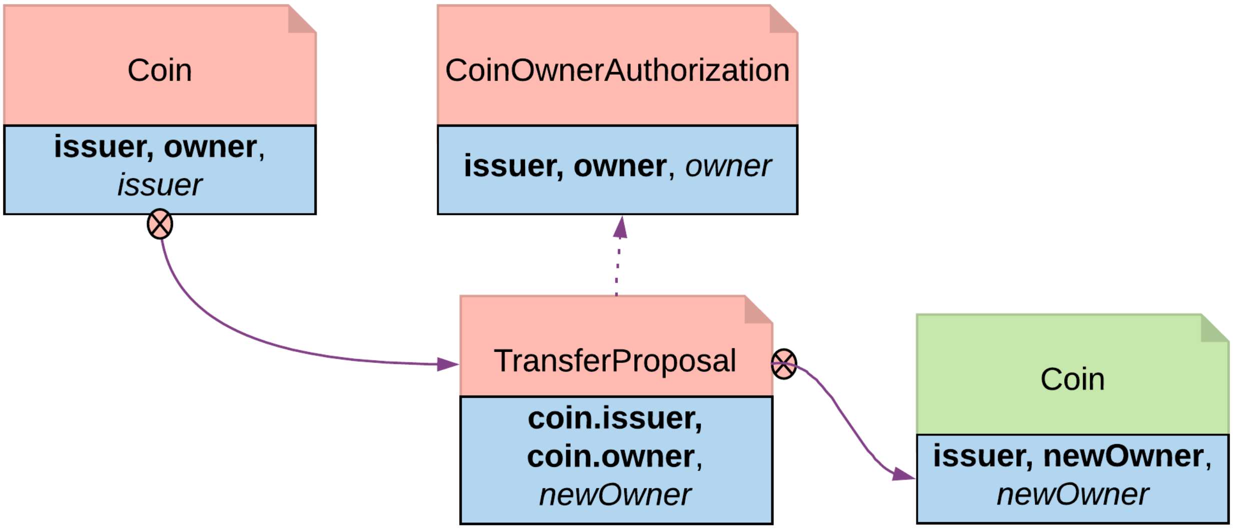

Coin
issuer, owner, *issuer*
CoinOwnerAuthorization
issuer, owner, *owner*
TransferProposal
coin.issuer, coin.owner, *newOwner*
Coin
issuer, newOwner, *newOwner*

Fig. 4: Authorization Diagram

2.5.5 Locking

The Locking pattern exhibits how to achieve locking safely and efficiently in DAML. Only the specified locking party can lock the asset through an active and authorized action. When a contract is locked, some or all choices specified on that contract may not be exercised.

2.5.5.1 Motivation

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

2.5.5.2 Implementation

There are three ways to achieve locking:

Locking by archiving

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

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

    controller owner can

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

    --a coin can only be archived by the issuer under the condition that
↪the issuer is the owner of the coin. This ensures the issuer cannot
↪archive coins at will.
    controller issuer can
      Archives
        : ()
        do assert (issuer == owner)
```

Archiving is a straightforward choice for locking because once a contract is archived, all choices on the contract become unavailable. Archiving can be done either through consuming choice or archiving contract.

Consuming choice

The steps below show how to use a consuming choice in the original contract to achieve locking:

Add a consuming choice, *Lock*, to the *Coin* template that creates a *LockedCoin*.

The controller party on the *Lock* may vary depending on business context. In this example, *owner* is a good choice.
The parameters to this choice are also subject to business use case. Normally, it should have at least locking terms (eg. lock expiry time) and a party authorized to unlock.

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

Create a *LockedCoin* to represent *Coin* in the locked state. *LockedCoin* has the following characteristics, all in order to be able to recreate the original *Coin*:

- The signatories are the same as the original contract.
- It has all data of *Coin*, either through having a *Coin* as a field, or by replicating all data of *Coin*.
- It has an *Unlock* choice to lift the lock.

```
template LockedCoin
  with
    coin: Coin
    maturity: Time
    locker: Party
  where
    signatory coin.issuer, coin.owner

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

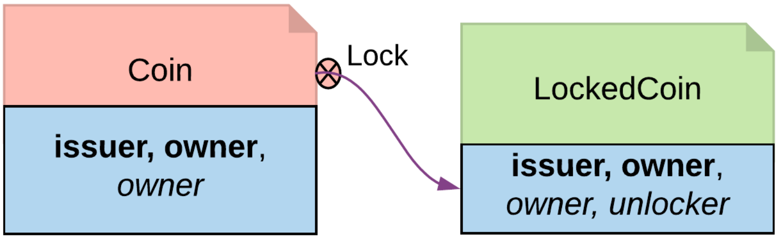


Fig. 5: Locking By Consuming Choice Diagram

Archiving contract

In the event that changing the original contract is not desirable and assuming the original contract already has an *Archive* choice, you can introduce another contract, *CoinCommitment*, to archive *Coin* and create *LockedCoin*.

Examine the controller party and archiving logic in the *Archives* choice on the *Coin* contract. A coin can only be archived by the issuer under the condition that the issuer is the owner of the coin. This ensures the issuer cannot archive any coin at will.

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

Since we need to call the *Archives* choice from *CoinCommitment*, its signatory has to be *Issuer*.

```
template CoinCommitment
  with
    owner: Party
    issuer: Party
    amount: Decimal
   where
    signatory issuer
```

The controller party and parameters on the *Lock* choice are the same as described in locking by consuming choice. The additional logic required is to transfer the asset to the issuer, and then explicitly call the *Archive* choice on the *Coin* contract.
Once a *Coin* is archived, the *Lock* choice creates a *LockedCoin* that represents *Coin* in locked state.

```
    controller owner can
      nonconsuming LockCoin
        : ContractId LockedCoin
        with coinCid: ContractId Coin
             maturity: Time
             locker: Party
        do
          inputCoin <- fetch coinCid
          assert (inputCoin.owner == owner && inputCoin.issuer == issuer &&
↪ inputCoin.amount == amount )
          --the original coin firstly transferred to issuer and then
↪archivaed
          prop <- exercise coinCid Transfer with newOwner = issuer
          do
            id <- exercise prop AcceptTransfer
            exercise id Archives
          --create a lockedCoin to represent the coin in locked state
          create LockedCoin with
            coin=inputCoin with owner; issuer; amount
            maturity; locker
```

Trade-offs

This pattern achieves locking in a fairly straightforward way. However, there are some tradeoffs.

Locking by archiving disables all choices on the original contract. Usually for consuming choices this is exactly what is required. But if a party needs to selectively lock only some choices, remaining active choices need to be replicated on the *LockedCoin* contract, which can lead to code duplication.
The choices on the original contract need to be altered for the lock choice to be added. If this contract is shared across multiple participants, it will require agreement from all involved.

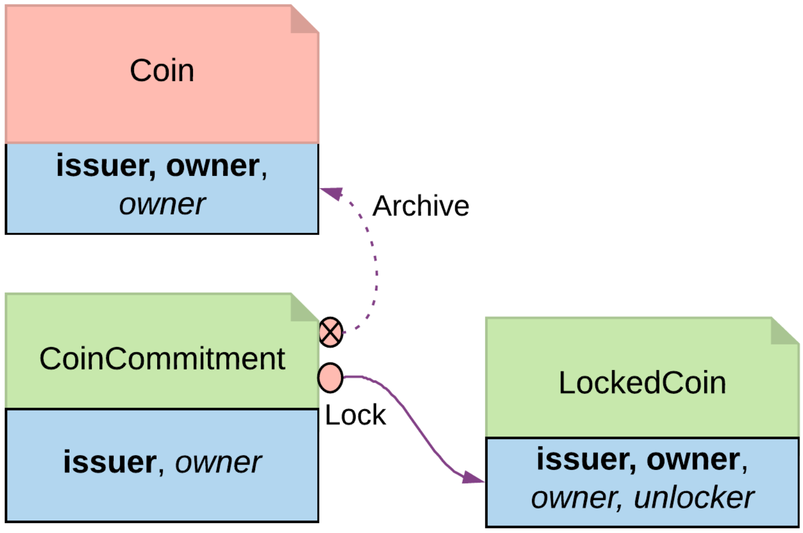


Fig. 6: Locking By Archiving Contract Diagram

Locking by state

The original *Coin* template is shown below. This is the basis on which to implement locking by state

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

    controller owner can

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

    --a coin can only be archived by the issuer under the condition that the issuer is the owner of the coin. This ensures the issuer cannot archive coins at will.
    controller issuer can
      Archives
        : ()
        do assert (issuer == owner)
```

In its original form, all choices are actionable as long as the contract is active. Locking by State requires introducing fields to track state. This allows for the creation of an active contract in two possible states: locked or unlocked. A DAML modeler can selectively make certain choices actionable only if the contract is in unlocked state. This effectively makes the asset lockable.

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

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

- Add a *locker* party to the template parameters.
- Define the states.
  - if owner == locker, the coin is unlocked
  - if owner != locker, the coin is in a locked state
- The contract state is checked on choices.
  - *Transfer* choice is only actionable if the coin is unlocked
  - *Lock* choice is only actionable if the coin is unlocked and a 3rd party locker is supplied
  - *Unlock* is available to the locker party only if the coin is locked

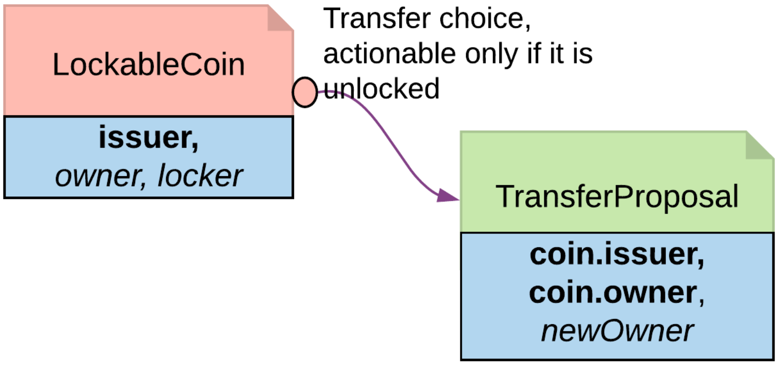
```
template LockableCoin
  with
    owner: Party
    issuer: Party
    amount: Decimal
    locker: Party
  where
    signatory issuer
    signatory owner

    ensure amount > 0.0

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

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

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

(continues on next page)

(continued from previous page)

```
          assert (locker /= owner)
          create this with locker = owner
```

Locking By State Diagram



Trade-offs

> It requires changes made to the original contract template. Furthermore you should need to change all choices intended to be locked.
> If locking and unlocking terms (e.g. lock triggering event, expiry time, etc) need to be added to the template parameters to track the state change, the template can get overloaded.

Locking by safekeeping

Safekeeping is a realistic way to model locking as it is a common practice in many industries. For example, during a real estate transaction, purchase funds are transferred to the sellers lawyer's escrow account after the contract is signed and before closing. To understand its implementation, review the original *Coin* template first.

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

    controller owner can

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

(continues on next page)

(continued from previous page)

```
    --a coin can only be archived by the issuer under the condition that the issuer is the owner of the coin. This ensures the issuer cannot archive coins at will.
    controller issuer can
      Archives
        : ()
        do assert (issuer == owner)
```

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

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

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

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

    controller coin.owner can
      Accept : LockResult
        with coinCid : ContractId Coin
        do
          inputCoin <- fetch coinCid
          assert (inputCoin == coin)
          tpCid <- exercise coinCid Transfer with newOwner = locker
          coinCid <- exercise tpCid AcceptTransfer
          lockCid <- create LockedCoinV2 with locker; maturity; coin
          return LockResult {coinCid; lockCid}
```

LockedCoinV2 represents *Coin* in the locked state. It is fairly similar to the *LockedCoin* described in Consuming choice. The additional logic is to transfer ownership from the locker back to the owner when *Unlock* or *Clawback* is called.

```
template LockedCoinV2
  with
    coin: Coin
    maturity: Time
    locker: Party
  where
    signatory locker, coin.owner

```

(continues on next page)

(continued from previous page)

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

    controller coin.owner can
      ClawbackV2
        : ContractId Coin
        with coinCid : ContractId Coin
        do
          currTime <- getTime
          assert (currTime >= maturity)
          inputCoin <- fetch coinCid
          assert (inputCoin == coin with owner=locker)
          tpCid <- exercise coinCid Transfer with newOwner = coin.owner
          exercise tpCid AcceptTransfer
```

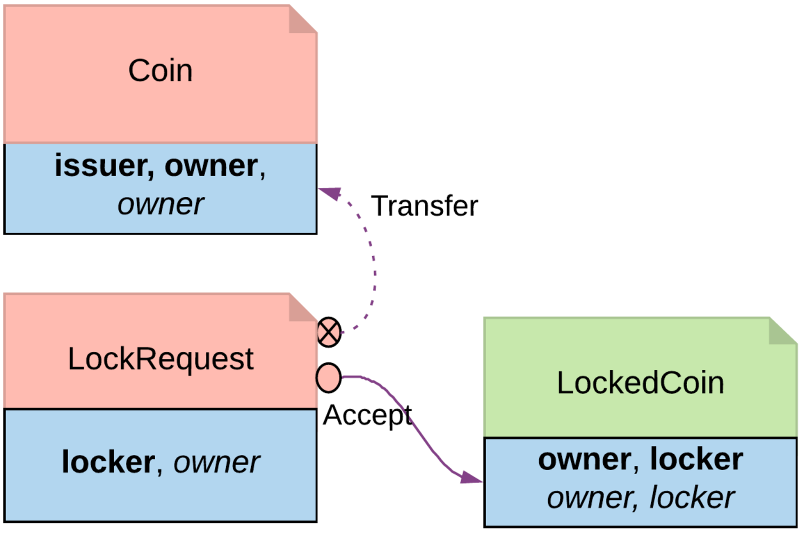
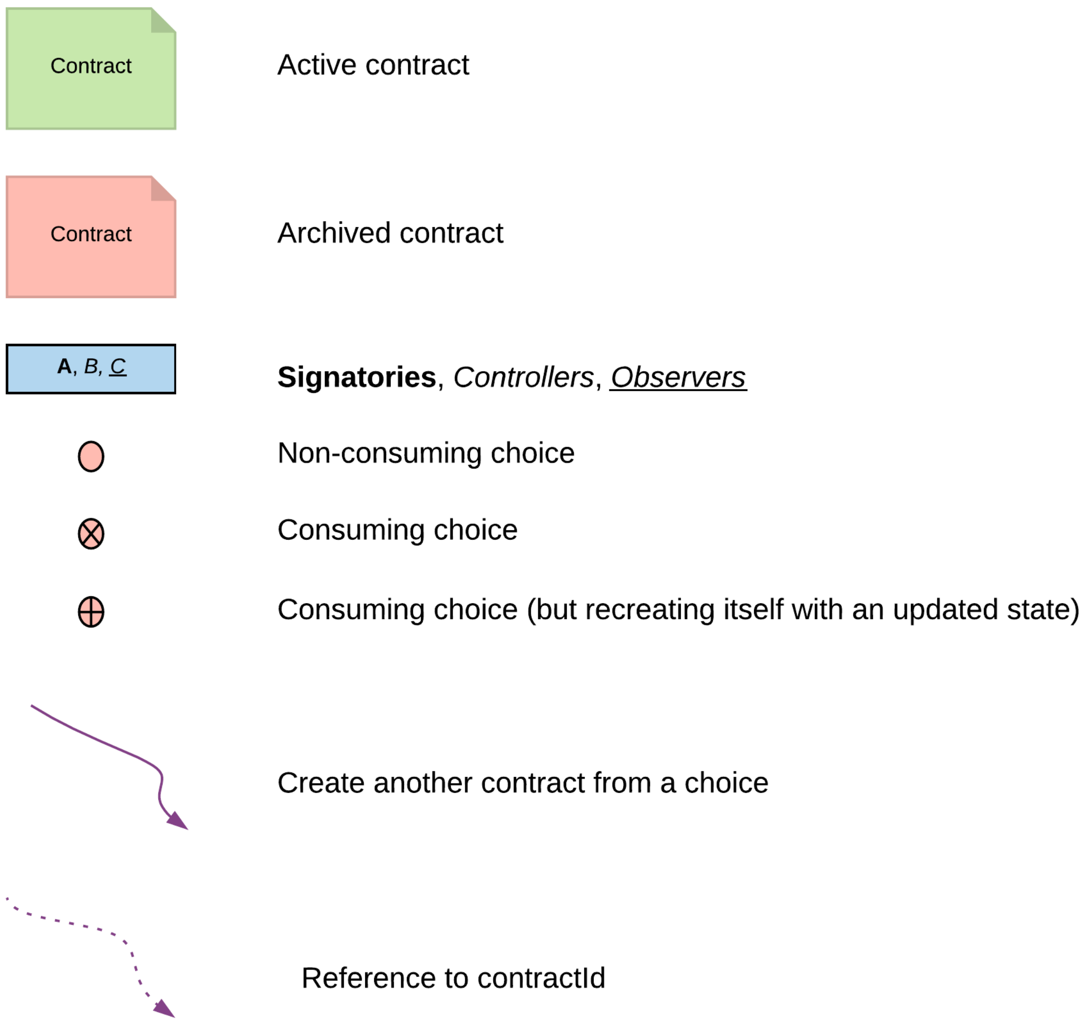


Fig. 7: Locking By Safekeeping Diagram

Trade-offs

Ownership transfer may give the locking party too much access on the locked asset. A rogue lawyer could run away with the funds. In a similar fashion, a malicious locker party could introduce code to transfer assets away while they are under their ownership.

2.5.6 Diagram legends

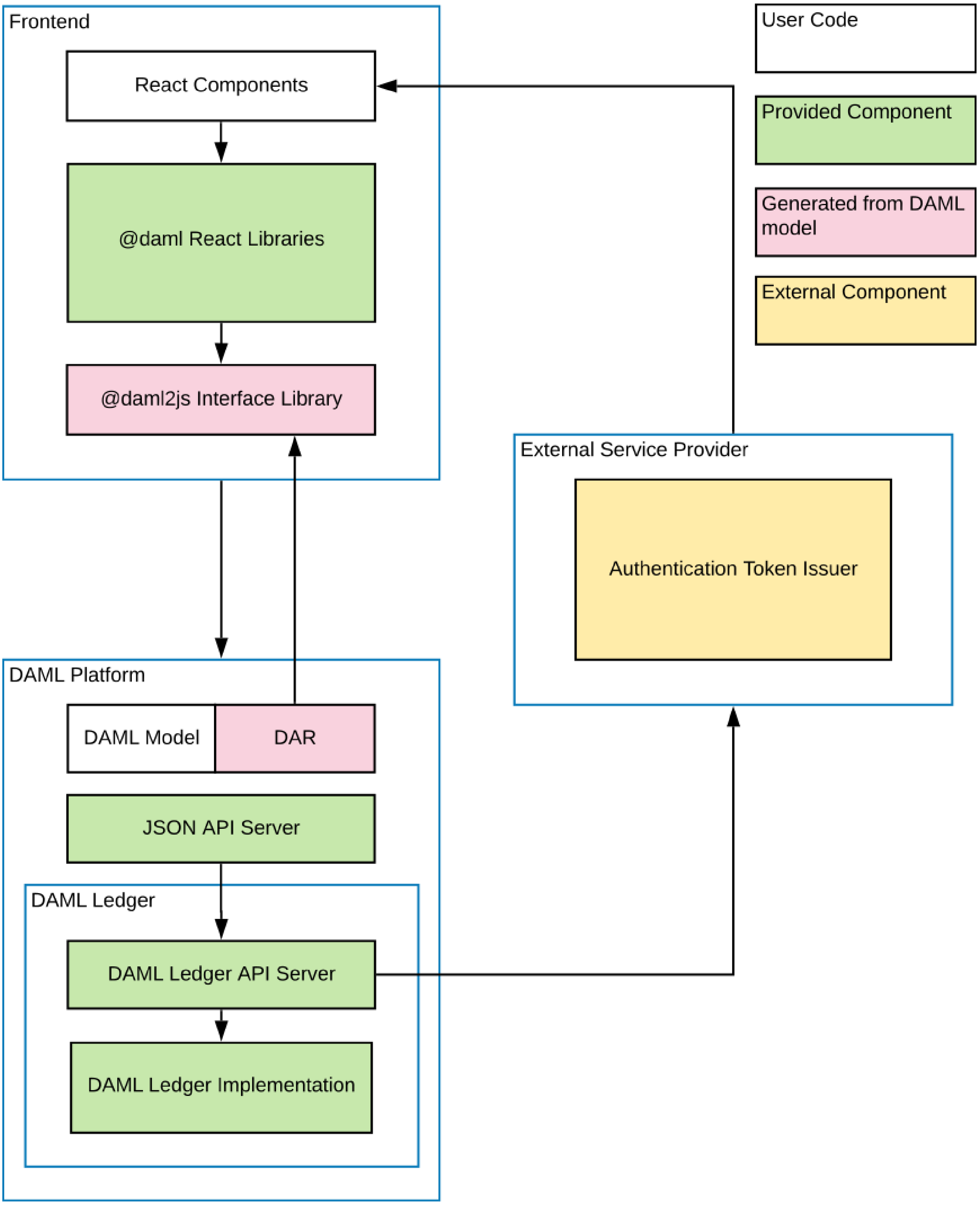

Contract
Active contract
Contract
Archived contract
A, *B*, C
Signatories, *Controllers*, Observers
Non-consuming choice
Consuming choice
Consuming choice (but recreating itself with an updated state)
Create another contract from a choice
Reference to contractId

Chapter 3

Building applications

3.1 Application architecture

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



The above image shows the recommended architecture. Of course there are many ways how you can change the architecture and technology stack to fit your needs, which we'll mention in the corresponding sections.

To get started quickly with the recommended application architecture clone the `create-daml-app` application template:

```
git clone https://github.com/digital-asset/create-daml-app
```

`create-daml-app` is a small, but fully functional demo application implementing the recommended architecture, providing you with an excellent starting point for your own application. It showcases

- using DAML React libraries
- quick iteration against the *DAML Ledger Sandbox*.
- authorization
- deploying your application in the cloud as a Docker container

3.1.1 Backend

The backend for your application can be any DAML ledger implementation running your DAR (*DAML Archive*) file.

We recommend using the *DAML JSON API* as an interface to your frontend. It is served by the HTTP JSON API server connected to the ledger API server. It provides simple HTTP endpoints to interact with the ledger via GET/POST requests. However, if you prefer, you can also use the *gRPC API* directly.

When you use the `create-daml-app` template application, you can start a local sandbox together with a JSON API server by running

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

in the root of the project. This is the most simple DAML ledger implementation. Once your application matures and becomes ready for production, the `daml deploy` command helps you deploy your frontend and DAML artifacts of your project to a production ledger. See *Deploying to DAML Ledgers* for an in depth manual for specific ledgers.

3.1.2 Frontend

We recommended building your frontend with the React framework. However, you can choose virtually any language for your frontend and interact with the ledger via *HTTP JSON* endpoints. In addition, we provide support libraries for *Java* and *Scala* and you can also interact with the *gRPC API* directly.

We provide two libraries to build your React frontend for a DAML application.

Name	Summary
@daml/react	React hooks to query/create/exercise DAML contracts
@daml/ledger	DAML ledger object to connect and directly submit commands to the ledger

You can install any of these libraries by running `yarn add <library>` in the `ui` directory of your project, e.g. `yarn add @daml/react`. Please explore the `create-daml-app` example project to see the usage of these libraries.

To make your life easy when interacting with the ledger, the DAML assistant can generate JavaScript libraries with TypeScript typings from the data types declared in the deployed DAR.

```
daml codegen js .daml/dist/<your-project-name.dar> -o daml.js
```

This command will generate a JavaScript library for each DALF in you DAR, containing metadata about types and templates in the DALF and TypeScript typings them. In `create-daml-app`, ui/package.json refers to these libraries via the "create-daml-app": "file:../daml.js/create-daml-app-0.1.0" entry in the dependencies field.

If you choose a different JavaScript based frontend framework, the packages @daml/ledger, @daml/types and the generated daml.js libraries provide you with the necessary code to connect and issue commands against your ledger.

3.1.3 Authorization

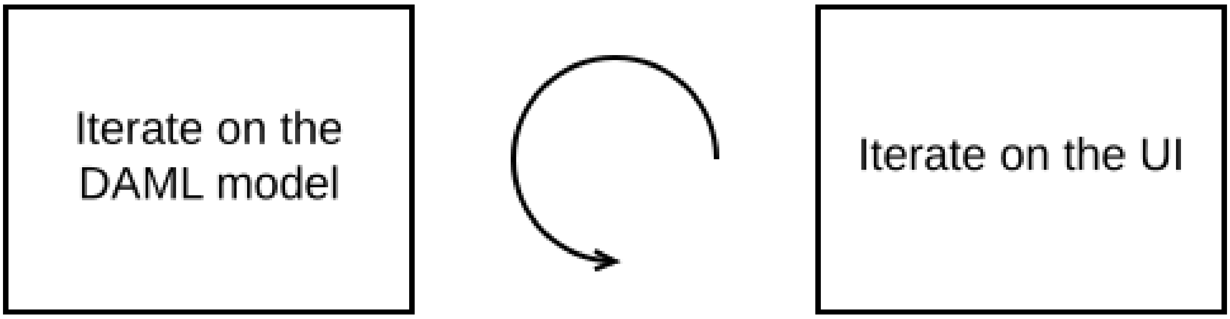
When you deploy your application to a production ledger, you need to authenticate the identities of your users.

DAML ledgers support a unified interface for authorization of commands. Some DAML ledgers, like for example https://projectdabl.com, offer integrated authentication and authorization, but you can also use an external service provider like https://auth0.com. The DAML react libraries support interfacing with a DAML ledger that validates authorization of incoming requests. Simply initialize your DamlLedger object with the token obtained by the respective token issuer. How authorization works and the form of the required tokens is described in the Authorization section.

3.1.4 Developer workflow

The DAML SDK enables a local development environment with fast iteration cycles. If you run daml-reload-on-change.sh of the create-daml-app, a local DAML sandbox ledger is started that is updated with your most recent DAML code on any change. Next, you can start your frontend in development mode by changing to your ui directory and run yarn start. This will reload your frontend whenever you make changes to it. You can add unit tests for your DAML models by writing *DAML scenarios*. These will also be reevaluated on change. A typical DAML developer workflow is to

1. Make a small change to your DAML data model
2. Optionally test your DAML code and with *scenarios*
3. Edit your React components to be aligned with changes made in DAML code
4. Extend the UI to make use of the newly introduced feature
5. Make further changes either to your DAML and/or React code until you're happy with what you've developed



3.1.4.1 Command deduplication

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

There are several things that can fail during this time window: the application can crash, the participant node can crash, messages can be lost on the network, or the ledger may be just slow to respond due to a high load.

If you want to make sure that a command is not executed twice, your application needs to robustly handle all the various failure scenarios. DAML ledgers provide a mechanism for *command deduplication* to help deal this problem.

For each command applications provide a command ID and an optional parameter that specifies the deduplication time. If the latter parameter is not specified in the command submission itself, the ledger will fall back to using the configured maximum deduplication time. The ledger will then guarantee that commands for the same submitting party and command ID will be ignored within the deduplication time window.

To use command deduplication, you should:

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

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

3.1.4.2 Failing over between Ledger API endpoints

Some DAML Ledgers support exposing multiple eventually consistent Ledger API endpoints where command deduplication works across these Ledger API endpoints. For example, these endpoints might be hosted by separate Ledger API servers that replicate the same data and host the same parties. Contact your ledger operator to find out whether this applies to your ledger.

Below we describe how you can build your application such that it can switch between such eventually consistent Ledger API endpoints to tolerate server failures. You can do this using the following two steps.

First, your application must keep track of the last ledger offset received from the *transaction service* or the *command completion service*. When switching to a new Ledger API endpoint, it must resume consumption of the transaction (tree) and/or the command completion streams starting from this last received offset.

Second, your application must retry on `OUT_OF_RANGE` errors (see gRPC status codes) received from a stream subscription – using an appropriate backoff strategy to avoid overloading the server. Such errors can be raised because of eventual consistency. The Ledger API endpoint that the application is newly subscribing to might be behind the endpoint that it subscribed to before the switch, and needs time to catch up. Thanks to eventual consistency this is guaranteed to happen at some point in the future.

Once the application successfully subscribes to its required streams on the new endpoint, it will resume normal operation.

3.1.4.3 Dealing with time

The DAML language contains a function *getTime* which returns the current time . The notion of time comes with a lot of problems in a distributed setting: different participants might run slightly different clocks, transactions would not be allowed to overtake each other during DAML interpretation, i.e., a long-running command could block all other commands, and many more.

To avoid such problems, DAML provides the following concept of *ledger time*:

> As part of command interpretation, each transaction is automatically assigned a *ledger time* by the participant server.
>
> All calls to `getTime` within a transaction return the *ledger time* assigned to that transaction.
>
> *Ledger time* is reasonably close to real time. To avoid transactions being rejected because the assigned *ledger time* does not match the ledger's system time exactly, DAML Ledgers define a tolerance interval around its system time. The system time is part of the ledger synchronization/consensus protocol, but is not known by the participant node at interpretation time. Transactions with a *ledger time* outside this tolerance interval will be rejected.
>
> *Ledger time* respects causal monotonicity: if a transaction `x` uses a contract created in another transaction `y`, transaction `x`s ledger time will be greater than or equal to the ledger time of the referenced transaction `y`.

Some commands might take a long time to process, and by the time the resulting transaction is about to be committed to the ledger, it might violate the condition that *ledger time* should be reasonably close to real time (even when considering the ledger's tolerance interval). To avoid such problems, applications can set the optional parameters *min_ledger_time_abs* or *min_ledger_time_rel* command parameters that specify (in absolute or relative terms) the minimal *ledger time* for the transaction. The ledger will then process the command, but wait with committing the resulting transaction until *ledger time* fits within the ledger's tolerance interval.

How is this used in practice?

> Be aware that `getTime` is only reasonably close to real time. Avoid DAML workflows that rely on very accurate time measurements or high frequency time changes.
>
> Set `min_ledger_time_abs` or `min_ledger_time_rel` if the duration of command interpretation and transmission is likely to take a long time relative to the tolerance interval set by the ledger.
>
> In some corner cases, the participant node may be unable to determine a suitable ledger time by itself. If you get an error that no ledger time could be found, check whether you have contention on any contract referenced by your command or whether the referenced contracts are sensitive to small changes of `getTime`.

For details, see Background concepts - time.

3.2 JavaScript Client Libraries

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

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

3.2.1 JavaScript Code Generator

The command `daml codegen js` generates JavaScript (and TypeScript) that can be used in conjunction with the JavaScript Client Libraries for interacting with a DAML ledger via the HTTP JSON API.

Inputs to the command are DAR files. Outputs are JavaScript packages with TypeScript typings containing metadata and types for all DAML packages included in the DAR files.

The generated packages use the library @daml/types.

3.2.1.1 Usage

In outline, the command to generate JavaScript and TypeScript typings from DAML is `daml codegen js -o OUTDIR DAR` where `DAR` is the path to a DAR file (generated via `daml build`) and `OUTDIR` is a directory where you want the artifacts to be written.

Here's a complete example on a project built from the standard skeleton template.

```
daml new my-proj --template skeleton # Create a new project based off the skeleton template
cd my-proj # Enter the newly created project directory
daml build  # Compile the project's DAML files into a DAR
daml codegen js -o daml.js .daml/dist/my-proj-0.0.1.dar # Generate JavaScript packages in the daml.js directory
```

On execution of these commands:

- The directory `my-proj/daml.js` contains generated JavaScript packages with TypeScript typings;
- The files are arranged into directories;
- One of those directories will be named my-proj-0.0.1 and will contain the definitions corresponding to the DAML files in the project;
- For example, `daml.js/my-proj-0.0.1/lib/index.js` provides access to the definitions for `daml/Main.daml`;
- The remaining directories correspond to modules of the DAML standard library;
- Those directories have numeric names (the names are hashes of the DAML-LF package they are derived from).

To get a quickstart idea of how to use what has been generated, you may wish to jump to the *Templates and choices* section and return to the reference material that follows as needed.

3.2.1.2 Primitive DAML types: @daml/types

To understand the TypeScript typings produced by the code generator, it is helpful to keep in mind this quick review of the TypeScript equivalents of the primitive DAML types provided by @daml/types.

Interfaces:

```
Template<T extends object, K = unknown>
Choice<T extends object, C, R, K = unknown>
```

Types:

DAML	TypeScript	TypeScript definition
()	Unit	{}
Bool	Bool	boolean
Int	Int	string
Decimal	Decimal	string
Numeric ν	Numeric	string
Text	Text	string
Time	Time	string
Party	Party	string
[τ]	List<τ>	τ[]
Date	Date	string
ContractId τ	ContractId<τ>	string
Optional τ	Optional<τ>	null | (null extends τ ? [] | [Exclude<τ, null>] : τ)
TextMap τ	TextMap<τ>	{ [key: string]: τ }
(τ_1, τ_2)	Tuple$_2$<τ_1, τ_2>	{_1: τ_1; _2: τ_2}

Note: The types given in the TypeScript column are defined in @daml/types.

Note: For *n*-tuples where *n* 3, representation is analogous with the pair case (the last line of the table).

Note: The TypeScript types `Time`, `Decimal`, `Numeric` and `Int` all alias to `string`. These choices relate to the avoidance of precision loss under serialization over the json-api.

Note: The TypeScript definition of type `Optional<τ>` in the above table might look complicated. It accounts for differences in the encoding of optional values when nested versus when they are not (i.e. top-level). For example, `null` and `"foo"` are two possible values of `Optional<Text>` whereas, `[]` and `["foo"]` are two possible values of type `Optional<Optional<Text>>` (`null` is another possible value, `[null]` is **not**).

3.2.1.3 DAML to TypeScript mappings

The mappings from DAML to TypeScript are best explained by example.

Records

In DAML, we might model a person like this.

```
data Person =
  Person with
```

(continues on next page)

(continued from previous page)

```
    name: Text
    party: Party
    age: Int
```

Given the above definition, the generated TypeScript code will be as follows.

```
type Person = {
  name: string;
  party: daml.Party;
  age: daml.Int;
}
```

Variants

This is a DAML type for a language of additive expressions.

```
data Expr a =
    Lit a
  | Var Text
  | Add (Expr a, Expr a)
```

In TypeScript, it is represented as a discriminated union.

```
type Expr<a> =
  |  { tag: 'Lit'; value: a }
  |  { tag: 'Var'; value: string }
  |  { tag: 'Add'; value: {_1: Expr<a>, _2: Expr<a>} }
```

Sum-of-products

Let's slightly modify the `Expr a` type of the last section into the following.

```
data Expr a =
    Lit a
  | Var Text
  | Add {lhs: Expr a, rhs: Expr a}
```

Compared to the earlier definition, the `Add` case is now in terms of a record with fields `lhs` and `rhs`. This renders in TypeScript like so.

```
type Expr<a> =
  |  { tag: 'Lit2'; value: a }
  |  { tag: 'Var2'; value: string }
  |  { tag: 'Add'; value: Expr.Add<a> }

namespace Expr {
  type Add<a> = {
    lhs: Expr<a>;
    rhs: Expr<a>;
```

(continues on next page)

(continued from previous page)

```
  }
}
```

The thing to note is how the definition of the Add case has given rise to a record type definition Expr.Add.

Enums

Given a DAML enumeration like this,

```
data Color = Red | Blue | Yellow
```

the generated TypeScript will consist of a type declaration and the definition of an associated companion object.

```
type Color = 'Red' | 'Blue' | 'Yellow'

const Color = {
  Red: 'Red',
  Blue: 'Blue',
  Yellow: 'Yellow',
  keys: ['Red','Blue','Yellow'],
} as const;
```

Templates and choices

Here is a DAML template of a basic 'IOU' contract.

```
template Iou
  with
    issuer: Party
    owner: Party
    currency: Text
    amount: Decimal
  where
    signatory issuer
    choice Transfer: ContractId Iou
      with
        newOwner: Party
      controller owner
      do
        create this with owner = newOwner
```

The daml codegen js command generates types for each of the choices defined on the template as well as the template itself.

```
type Transfer = {
  newOwner: daml.Party;
}

```

(continues on next page)

(continued from previous page)

```
type Iou = {
  issuer: daml.Party;
  owner: daml.Party;
  currency: string;
  amount: daml.Numeric;
}
```

Each template results in the generation of a companion object. Here, is a schematic of the one generated from the `Iou` template[2].

```
const Iou: daml.Template<Iou, undefined> & {
  Archive: daml.Choice<Iou, DA_Internal_Template.Archive, {}, undefined>;
  Transfer: daml.Choice<Iou, Transfer, daml.ContractId<Iou>, undefined>;
} = {
  /* ... */
}
```

The exact details of these companion objects are not important - think of them as representing metadata .

What **is** important is the use of the companion objects when creating contracts and exercising choices using the @daml/ledger package. The following code snippet demonstrates their usage.

```
1  import Ledger from  '@daml/ledger';
2  import {Iou, Transfer} from /* ... */;
3
4  const ledger = new Ledger(/* ... */);
5
6  // Contract creation; Bank issues Alice a USD $1MM IOU.
7
8  const iouDetails: Iou = {
9    issuer: 'Chase',
10   owner: 'Alice',
11   currency: 'USD',
12   amount: 1000000.0,
13 };
14 const aliceIouCreateEvent = await ledger.create(Iou, iouDetails);
15 const aliceIouContractId = aliceIouCreateEvent.contractId;
16
17 // Choice execution; Alice transfers ownership of the IOU to Bob.
18
19 const transferDetails: Transfer = {
20   newOwner: 'Bob',
21 }
22 const [bobIouContractId, _] = await ledger.exercise(Transfer, aliceIouContractId, transferDetails);
```

Observe on line 14, the first argument to `create` is the `Iou` companion object and on line 22, the first argument to `exercise` is the `Transfer` companion object.

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

3.2.2 @daml/react

@daml/react documentation

3.2.3 @daml/ledger

@daml/ledger documentation

3.2.4 @daml/types

@daml/types documentation

3.3 HTTP JSON API Service

The **JSON API** provides a significantly simpler way to interact with a ledger than *the Ledger API* by providing *basic active contract set functionality*:

- creating contracts,
- exercising choices on contracts,
- querying the current active contract set, and
- retrieving all known parties.

The goal of this API is to get your distributed ledger application up and running quickly, so we have deliberately excluded complicating concerns including, but not limited to:

- inspecting transactions,
- asynchronous submit/completion workflows,
- temporal queries (e.g. active contracts *as of a certain time*), and
- ledger metaprogramming (e.g. retrieving packages and templates).

For these and other features, use *the Ledger API* instead.

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

3.3.1 DAML-LF JSON Encoding

We describe how to decode and encode DAML-LF values as JSON. For each DAML-LF type we explain what JSON inputs we accept (decoding), and what JSON output we produce (encoding).

The output format is parameterized by two flags:

```
encodeDecimalAsString: boolean
encodeInt64AsString: boolean
```

The suggested defaults for both of these flags is false. If the intended recipient is written in JavaScript, however, note that the JavaScript data model will decode these as numbers, discarding data in some cases; encode-as-String avoids this, as mentioned with respect to `JSON.parse` below.

Note that throughout the document the decoding is type-directed. In other words, the same JSON value can correspond to many DAML-LF values, and the expected DAML-LF type is needed to decide which one.

3.3.1.1 ContractId

Contract ids are expressed as their string representation:

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

3.3.1.2 Decimal

Input

Decimals can be expressed as JSON numbers or as JSON strings. JSON strings are accepted using the same format that JSON accepts, and treated them as the equivalent JSON number:

```
-?(?:0|[1-9]\d*)(?:\.\d+)?(?:[eE][+-]?\d+)?
```

Note that JSON numbers would be enough to represent all Decimals. However, we also accept strings because in many languages (most notably JavaScript) use IEEE Doubles to express JSON numbers, and IEEE Doubles cannot express DAML-LF Decimals correctly. Therefore, we also accept strings so that JavaScript users can use them to specify Decimals that do not fit in IEEE Doubles.

Numbers must be within the bounds of Decimal, [–(10^{38}–1) 10^{10}, (10^{38}–1) 10^{10}]. Numbers outside those bounds will be rejected. Numbers inside the bounds will always be accepted, using banker's rounding to fit them within the precision supported by Decimal.

A few valid examples:

```
42 --> 42
42.0 --> 42
"42" --> 42
9999999999999999999999999999.9999999999 -->
    9999999999999999999999999999.9999999999
-42 --> -42
"-42" --> -42
0 --> 0
-0 --> 0
0.30000000000000004 --> 0.3
2e3 --> 2000
```

A few invalid examples:

```
"  42  "
"blah"
99999999999999999999999999990
+42
```

Output

If encodeDecimalAsString is set, decimals are encoded as strings, using the format `-?[0-9]{1,28}(\.[0-9]{1,10})?`. If encodeDecimalAsString is not set, they are encoded as JSON numbers, also using the format `-?[0-9]{1,28}(\.[0-9]{1,10})?`.

Note that the flag encodeDecimalAsString is useful because it lets JavaScript consumers consume Decimals safely with the standard JSON.parse.

3.3.1.3 Int64

Input

Int64, much like Decimal, can be represented as JSON numbers and as strings, with the string representation being `[+-]?[0-9]+`. The numbers must fall within [-9223372036854775808, 9223372036854775807]. Moreover, if represented as JSON numbers, they must have no fractional part.

A few valid examples:

```
42
"+42"
-42
0
-0
9223372036854775807
"9223372036854775807"
-9223372036854775808
"-9223372036854775808"
```

A few invalid examples:

```
42.3
+42
9223372036854775808
-9223372036854775809
"garbage"
"   42 "
```

Output

If encodeInt64AsString is set, Int64s are encoded as strings, using the format `-?[0-9]+`. If encodeInt64AsString is not set, they are encoded as JSON numbers, also using the format `-?[0-9]+`.

Note that the flag encodeInt64AsString is useful because it lets JavaScript consumers consume Int64s safely with the standard `JSON.parse`.

3.3.1.4 Timestamp

Input

Timestamps are represented as ISO 8601 strings, rendered using the format `yyyy-mm-ddThh:mm:ss.ssssssZ`:

```
1990-11-09T04:30:23.123456Z
9999-12-31T23:59:59.999999Z
```

Parsing is a little bit more flexible and uses the format `yyyy-mm-ddThh:mm:ss(\.s+)?Z`, i.e. it's OK to omit the microsecond part partially or entirely, or have more than 6 decimals. Sub-second data beyond microseconds will be dropped. The UTC timezone designator must be included. The rationale behind the inclusion of the timezone designator is minimizing the risk that users pass in local times. Valid examples:

```
1990-11-09T04:30:23.1234569Z
1990-11-09T04:30:23Z
1990-11-09T04:30:23.123Z
0001-01-01T00:00:00Z
9999-12-31T23:59:59.999999Z
```

The timestamp must be between the bounds specified by DAML-LF and ISO 8601, [0001-01-01T00:00:00Z, 9999-12-31T23:59:59.999999Z].

JavaScript

```
> new Date().toISOString()
'2019-06-18T08:59:34.191Z'
```

Python

```
>>> datetime.datetime.utcnow().isoformat() + 'Z'
'2019-06-18T08:59:08.392764Z'
```

Java

```
import java.time.Instant;
class Main {
    public static void main(String[] args) {
        Instant instant = Instant.now();
        // prints 2019-06-18T09:02:16.652Z
        System.out.println(instant.toString());
    }
}
```

Output

Timestamps are encoded as ISO 8601 strings, rendered using the format `yyyy-mm-ddThh:mm:ss[.ssssss]Z`.

The sub-second part will be formatted as follows:

> If no sub-second part is present in the timestamp (i.e. the timestamp represents whole seconds), the sub-second part will be omitted entirely;
> If the sub-second part does not go beyond milliseconds, the sub-second part will be up to milliseconds, padding with trailing 0s if necessary;
> Otherwise, the sub-second part will be up to microseconds, padding with trailing 0s if necessary.

In other words, the encoded timestamp will either have no sub-second part, a sub-second part of length 3, or a sub-second part of length 6.

3.3.1.5 Party

Represented using their string representation, without any additional quotes:

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

3.3.1.6 Unit

Represented as empty object `{}`. Note that in JavaScript `{} !== {}`; however, `null` would be ambiguous; for the type `Optional Unit`, `null` decodes to `None`, but `{}` decodes to `Some ()`.

Additionally, we think that this is the least confusing encoding for Unit since unit is conceptually an empty record. We do not want to imply that Unit is used similarly to null in JavaScript or None in Python.

3.3.1.7 Date

Represented as an ISO 8601 date rendered using the format `yyyy-mm-dd`:

```
2019-06-18
9999-12-31
0001-01-01
```

The dates must be between the bounds specified by DAML-LF and ISO 8601, [0001-01-01, 9999-99-99].

3.3.1.8 Text

Represented as strings.

3.3.1.9 Bool

Represented as booleans.

3.3.1.10 Record

Input

Records can be represented in two ways. As objects:

```
{ f₁: v₁, ..., f□: v□ }
```

And as arrays:

```
[ v₁, ..., v□ ]
```

Note that DAML-LF record fields are ordered. So if we have

```
record Foo = {f1: Int64, f2: Bool}
```

when representing the record as an array the user must specify the fields in order:

```
[42, true]
```

The motivation for the array format for records is to allow specifying tuple types closer to what it looks like in DAML. Note that a DAML tuple, i.e. (42, True), will be compiled to a DAML-LF record `Tuple2 { _1 = 42, _2 = True }`.

Output

Records are always encoded as objects.

3.3.1.11 List

Lists are represented as

$$[v_1, \ldots, v_□]$$

3.3.1.12 TextMap

TextMaps are represented as objects:

$$\{ k_1: v_1, \ldots, k_□: v_□ \}$$

3.3.1.13 GenMap

GenMaps are represented as lists of pairs:

$$[[k_1, v_1], [k_□, v_□]]$$

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

3.3.1.14 Optional

Input

Optionals are encoded using `null` if the value is None, and with the value itself if it's Some. However, this alone does not let us encode nested optionals unambiguously. Therefore, nested Optionals are encoded using an empty list for None, and a list with one element for Some. Note that after the top-level Optional, all the nested ones must be represented using the list notation.

A few examples, using the form

```
JSON  -->  DAML-LF  :  Expected DAML-LF type
```

to make clear what the target DAML-LF type is:

```
null     -->  None                    : Optional Int64
null     -->  None                    : Optional (Optional Int64)
42       -->  Some 42                 : Optional Int64
[]       -->  Some None               : Optional (Optional Int64)
[42]     -->  Some (Some 42)          : Optional (Optional Int64)
[[]]     -->  Some (Some None)        : Optional (Optional (Optional Int64))
[[42]]   -->  Some (Some (Some 42))   : Optional (Optional (Optional Int64))
...
```

Finally, if Optional values appear in records, they can be omitted to represent None. Given DAML-LF types

```
record Depth1 = { foo: Optional Int64 }
record Depth2 = { foo: Optional (Optional Int64) }
```

We have

```
{ }              -->  Depth1 { foo: None }              :  Depth1
{ }              -->  Depth2 { foo: None }              :  Depth2
{ foo: 42 }      -->  Depth1 { foo: Some 42 }           :  Depth1
{ foo: [42] }    -->  Depth2 { foo: Some (Some 42) }    :  Depth2
{ foo: null }    -->  Depth1 { foo: None }              :  Depth1
{ foo: null }    -->  Depth2 { foo: None }              :  Depth2
{ foo: [] }      -->  Depth2 { foo: Some None }         :  Depth2
```

Note that the shortcut for records and Optional fields does not apply to Map (which are also represented as objects), since Map relies on absence of key to determine what keys are present in the Map to begin with. Nor does it apply to the [f$_1$, ..., f□] record form; Depth1 None in the array notation must be written as [null].

Type variables may appear in the DAML-LF language, but are always resolved before deciding on a JSON encoding. So, for example, even though Oa doesn't appear to contain a nested Optional, it may contain a nested Optional by virtue of substituting the type variable a:

```
record Oa a = { foo: Optional a }

{ foo: 42 }     -->  Oa { foo: Some 42 }         : Oa Int
{ }             -->  Oa { foo: None }            : Oa Int
{ foo: [] }     -->  Oa { foo: Some None }       : Oa (Optional Int)
{ foo: [42] }   -->  Oa { foo: Some (Some 42) }  : Oa (Optional Int)
```

In other words, the correct JSON encoding for any LF value is the one you get when you have eliminated all type variables.

Output

Encoded as described above, never applying the shortcut for None record fields; e.g. { foo: None } will always encode as { foo: null }.

3.3.1.15 Variant

Variants are expressed as

```
{ tag: constructor, value: argument }
```

For example, if we have

```
variant Foo = Bar Int64 | Baz Unit | Quux (Optional Int64)
```

These are all valid JSON encodings for values of type Foo:

```
{"tag": "Bar", "value": 42}
{"tag": "Baz", "value": {}}
{"tag": "Quux", "value": null}
{"tag": "Quux", "value": 42}
```

Note that DAML data types with named fields are compiled by factoring out the record. So for example if we have

```
data Foo = Bar {f1: Int64, f2: Bool} | Baz
```

We'll get in DAML-LF

```
record Foo.Bar = {f1: Int64, f2: Bool}
variant Foo = Bar Foo.Bar | Baz Unit
```

and then, from JSON

```
{"tag": "Bar", "value": {"f1": 42, "f2": true}}
{"tag": "Baz", "value": {}}
```

This can be encoded and used in TypeScript, including exhaustiveness checking; see a type refinement example.

3.3.1.16 Enum

Enums are represented as strings. So if we have

```
enum Foo = Bar | Baz
```

There are exactly two valid JSON values for Foo, `"Bar"` and `"Baz"`.

3.3.2 Query language

The body of `POST /v1/query` looks like so:

```
{
    "templateIds": [...template IDs...],
    "query": {...query elements...}
}
```

The elements of that query are defined here.

3.3.2.1 Fallback rule

Unless otherwise required by one of the other rules below or to follow, values are interpreted according to *DAML-LF JSON Encoding*, and compared for equality.

All types are supported by this simple equality comparison except:

lists
textmaps
genmaps

3.3.2.2 Simple equality

Match records having at least all the (potentially nested) keys expressed in the query. The result record may contain additional properties.

Example: `{ person: { name: "Bob" }, city: "London" }`

Match: `{ person: { name: "Bob", dob: "1956-06-21" }, city: "London", createdAt: "2019-04-30T12:34:12Z" }`
No match: `{ person: { name: "Bob" }, city: "Zurich" }`

Typecheck failure: `{ person: { name: ["Bob", "Sue"] }, city: "London" }`

A JSON object, when considered with a record type, is always interpreted as a field equality query. Its type context is thus mutually exclusive with comparison queries.

3.3.2.3 Comparison query

Match values on comparison operators for int64, numeric, text, date, and time values. Instead of a value, a key can be an object with one or more operators: `{ <op>: value }` where `<op>` can be:

"`%lt`" for less than
"`%gt`" for greater than
"`%lte`" for less than or equal to
"`%gte`" for greater than or equal to

"`%lt`" and "`%lte`" may not be used at the same time, and likewise with "`%gt`" and "`%gte`", but all other combinations are allowed.

Example: `{ "person" { "dob": { "%lt": "2000-01-01", "%gte": "1980-01-01" } } }`

Match: `{ person: { dob: "1986-06-21" } }`
No match: `{ person: { dob: "1976-06-21" } }`
No match: `{ person: { dob: "2006-06-21" } }`

These operators cannot occur in objects interpreted in a record context, nor may other keys than these four operators occur where they are legal, so there is no ambiguity with field equality.

3.3.2.4 Appendix: Type-aware queries

This section is non-normative.

This is not a *JSON* query language, it is a *DAML-LF* query language. So, while we could theoretically treat queries (where not otherwise interpreted by the may contain additional properties rule above) without concern for what LF type (i.e. template) we're considering, we *will not* do so.

Consider the subquery `{"foo": "bar"}`. This query conforms to types, among an unbounded number of others:

```
record A □ { foo : Text }
record B □ { foo : Optional Text }
variant C □ foo : Party | bar : Unit

// NB: LF does not require any particular case for VariantCon or Field;
// these are perfectly legal types in DAML-LF packages
```

In the cases of `A` and `B`, "`foo`" is part of the query language, and only "`bar`" is treated as an LF value; in the case of `C`, the whole query is treated as an LF value. The wide variety of ambiguous interpretations about what elements are interpreted, and what elements treated as literal, and *how* those elements are interpreted or compared, would preclude many techniques for efficient query compilation and LF value representation that we might otherwise consider.

Additionally, it would be extremely easy to overlook unintended meanings of queries when writing them, and impossible in many cases to suppress those unintended meanings within the query language. For example, there is no way that the above query could be written to match `A` but never `C`.

For these reasons, as with LF value input via JSON, queries written in JSON are also always interpreted with respect to some specified LF types (e.g. template IDs). For example:

```
{
    "templateIds": ["Foo:A", "Foo:B", "Foo:C"],
    "query": {"foo": "bar"}
}
```

will treat "foo" as a field equality query for A and B, and (supposing templates' associated data types were permitted to be variants, which they are not, but for the sake of argument) as a whole value equality query for C.

The above Typecheck failure happens because there is no LF type to which both "Bob" and ["Bob", "Sue"] conform; this would be caught when interpreting the query, before considering any contracts.

3.3.3 Running the JSON API

3.3.3.1 Start a DAML Ledger

You can run the JSON API alongside any ledger exposing the gRPC Ledger API you want. If you don't have an existing ledger, you can start an in-memory sandbox:

```
daml new my_project --template quickstart-java
cd my_project
daml build
daml sandbox --wall-clock-time --ledgerid MyLedger ./.daml/dist/quickstart-
↪0.0.1.dar
```

3.3.3.2 Start the HTTP JSON API Service

Basic

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

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

This will start the JSON API on port 7575 and connect it to a ledger running on `localhost:6865`.

Note: Your JSON API service should never be exposed to the internet. When running in production the JSON API should be behind a reverse proxy, such as via NGINX.

With Query Store

To improve the performance of the JSON API you can configure it to use a PostgreSQL backend as a cache. This is particularly beneficial if your ACS changes only very little (compared to the whole ACS size) between queries. Note that the PostgreSQL backend acts purely as a cache. It is save to reinitialize the database at any time.

To enable the PostgreSQL backend you can use the `--query-store-jdbc-config` flag, an example of which is below.

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

```
daml json-api --ledger-host localhost --ledger-port 6865 --http-port 7575 \
--query-store-jdbc-config "driver=org.postgresql.Driver,
↪url=jdbc:postgresql://localhost:5432/test?&ssl=true,user=postgres,
↪password=password,createSchema=false"
```

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

3.3.3.3 Access Tokens

The JSON API essentially performs two separate tasks:

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

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

Internal Access Token

This access token is used exclusively by the JSON API service for maintaining the internal list of known packages and templates that it gets from the Ledger API.

Note: At no point should this access token be provided to an end user, these are for internal use only.

Every access token is different and will depend on your specific ledger operator's requirements. The JSON API server requires no access to party-specific data, only access to the ledger identity and package services. These services are public meaning that you need a valid token to access them but no party-specific claims nor an admin claim. Please refer to your ledger operator's documentation to find out how to get these tokens from your ledger operator.

Once you have retrieved your access token, you can provide it to the JSON API by storing it in a file and starting `daml json-api` with the flag `--access-token-file /path/to/your/token.file`.

If the token cannot be read from the provided path or the Ledger API reports an authentication error (for example due to token expiration), the JSON API will report the error via logging.

Note: If the token file is updated with a new token it will be picked up at the next attempt to send a request. You can use this to handle cases where an old token expires without restarting your JSON API service.

Party-specific Access Tokens

Party-specific requests, i.e., command submissions and queries, require a JWT with some additional restrictions compared to the the format *described in the Token Payload section here*. The set of parties listed in *actAs* and *readAs* must contain exactly one party. In addition to that, the application id and ledger id are mandatory. HTTP requests pass the token in a header, while WebSocket requests pass the token in a subprotocol.

Note: While the JSON API receives the token it doesn't validate it itself. Upon receiving a token it will pass it, and all data contained within the request, on to the Ledger API's AuthService which will then determine if the token is valid and authorized. However, the JSON API does decode the token to extract the ledger id, application id and party so it requires that you use the JWT format documented below.

For a ledger without authorization, e.g., the default configuration of DAML Sandbox, you can use https://jwt.io (or the JWT library of your choice) to generate your token. You can use an arbitrary secret here. The default header is fine. Under Payload , fill in:

```
{
  "https://daml.com/ledger-api": {
    "ledgerId": "MyLedger",
    "applicationId": "foobar",
    "actAs": ["Alice"]
  }
}
```

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

Note: The value of `applicationId` will be used for commands submitted using that token.

The value for `actAs` is specified as a list and you provide it with the party that you want to use. Such as the example which uses `Alice` for a party. Each request can only be for one party. For example you couldn't have `actAs` defined as `["Alice", "Bob"]`.

The party should reference an already allocated party.

Note: As mentioned above the JSON API does not validate tokens so if your ledger runs without authorization you can use an arbitrary secret.

Then the Encoded box should have your **token**, ready for passing to the service as described in the following sections.

Alternatively, here are two tokens you can use for testing:

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

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJodHRwczovL2RhbWwuY29tL2xlZGdlci1hcGkiOnsibGVkZ2VySWQiOiJNeUxlZGdlciIsImFwcGxpY
↪34zzF_fbWv7p60r5s1kKzwndvGdsJDX-W4Xhm4oVdpk
```

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

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

Auth via HTTP

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

Example:

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

Auth via WebSockets

WebSocket clients support a subprotocols argument (sometimes simply called protocols); this is usually in a list form but occasionally in comma-separated form. Check documentation for your WebSocket library of choice for details.

For HTTP JSON requests, you must pass two subprotocols:

```
daml.ws.auth
jwt.token.paste-jwt-here
```

Example:

```
jwt.token.eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJodHRwczovL2RhbWwuY29tL2xlZGdlci1hcGkiOnsibGVkZ2VySWQiOiJNeUxlZGdlciIsImFwcGxpY
↪34zzF_fbWv7p60r5s1kKzwndvGdsJDX-W4Xhm4oVdpk``
```

3.3.4 HTTP Status Codes

The **JSON API** reports errors using standard HTTP status codes. It divides HTTP status codes into 3 groups indicating:

1. success (200)
2. failure due to a client-side problem (400, 401, 404)
3. failure due to a server-side problem (500)

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

200 - OK
400 - Bad Request (Client Error)
401 - Unauthorized, authentication required

404 - Not Found
500 - Internal Server Error

If a client's HTTP GET or POST request reaches an API endpoint, the corresponding response will always contain a JSON object with a `status` field, either an `errors` or `result` field and an optional `warnings`:

```
{
    "status": <400 | 401 | 404 | 500>,
    "errors": <JSON array of strings>, | "result": <JSON object or array>,
    ["warnings": <JSON object> ]
}
```

Where:

`status` – a JSON number which matches the HTTP response status code returned in the HTTP header,
`errors` – a JSON array of strings, each string represents one error,
`result` – a JSON object or JSON array, representing one or many results,
`warnings` – an optional field with a JSON object, representing one or many warnings.

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

3.3.4.1 Successful response, HTTP status: 200 OK

Content-Type: `application/json`
Content:

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

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

Content-Type: `application/json`
Content:

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

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

Content-Type: `application/json`
Content:

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

3.3.4.4 Examples

Result with JSON Object without Warnings:

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

Result with JSON Array and Warnings:

```
{"status": 200, "result": [...], "warnings": {"unknownTemplateIds": [
↪"UnknownModule:UnknownEntity"]}}
```

Bad Request Error:

```
{"status": 400, "errors": ["JSON parser error: Unexpected character 'f' at
↪input index 27 (line 1, position 28)"]}
```

Bad Request Error with Warnings:

```
{"status":400, "errors":["Cannot resolve any template ID from request"],
↪"warnings":{"unknownTemplateIds":["XXX:YYY","AAA:BBB"]}}
```

Authentication Error:

```
{"status": 401, "errors": ["Authentication Required"]}
```

Not Found Error:

```
{"status": 404, "errors": ["HttpMethod(POST), uri: http://localhost:7575/
↪v1/query1"]}
```

Internal Server Error:

```
{"status": 500, "errors": ["Cannot initialize Ledger API"]}
```

3.3.5 Create a new Contract

To create an instance of an `Iou` contract from the *Quickstart guide*:

```
template Iou
  with
    issuer : Party
    owner : Party
    currency : Text
    amount : Decimal
    observers : [Party]
```

3.3.5.1 HTTP Request

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

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

Where:

`templateId` is the contract template identifier, which can be formatted as either:

- `"<package ID>:<module>:<entity>"` or
- `"<module>:<entity>"` if contract template can be uniquely identified by its module and entity name.

`payload` field contains contract fields as defined in the DAML template and formatted according to *DAML-LF JSON Encoding*.

3.3.5.2 HTTP Response

Content-Type: `application/json`

Content:

```
{
    "status": 200,
    "result": {
        "observers": [],
        "agreementText": "",
        "payload": {
            "observers": [],
            "issuer": "Alice",
            "amount": "999.99",
            "currency": "USD",
            "owner": "Alice"
        },
        "signatories": [
            "Alice"
        ],
        "contractId": "#124:0",
        "templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:Iou
↪"
    }
}
```

Where:

`status` field matches the HTTP response status code returned in the HTTP header,

`result` field contains created contract details. Keep in mind that `templateId` in the **JSON API** response is always fully qualified (always contains package ID).

3.3.6 Creating a Contract with a Command ID

When creating a new contract you may specify an optional `meta` field. This allows you to control the *commandId* used when submitting a commend to the ledger.

Note: You cannot currently use `commandIds` anywhere else in the JSON API, but you can use it for observing the results of its commands outside the JSON API in logs or via the Ledger API's *Command Services*

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

Where:

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

3.3.7 Exercise by Contract ID

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

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

3.3.7.1 HTTP Request

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

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

(continues on next page)

(continued from previous page)

```
    }
}
```

Where:

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

3.3.7.2 HTTP Response

Content-Type: application/json
Content:

```
{
    "status": 200,
    "result": {
        "exerciseResult": "#201:1",
        "events": [
            {
                "archived": {
                    "contractId": "#124:0",
                    "templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:Iou
↪"
                }
            },
            {
                "created": {
                    "observers": [],
                    "agreementText": "",
                    "payload": {
                        "iou": {
                            "observers": [],
                            "issuer": "Alice",
                            "amount": "999.99",
                            "currency": "USD",
                            "owner": "Alice"
                        },
                        "newOwner": "Alice"
                    },
                    "signatories": [
                        "Alice"
                    ],
                    "contractId": "#201:1",
                    "templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:IouTransfer
↪"
                }
```

(continues on next page)

(continued from previous page)

```
            }
        ]
    }
}
```

Where:

- `status` field matches the HTTP response status code returned in the HTTP header,
- `result` field contains contract choice execution details:
  - `exerciseResult` field contains the return value of the exercised contract choice,
  - `events` contains an array of contracts that were archived and created as part of the choice execution. The array may contain: **zero or many** `{"archived": {...}}` and **zero or many** `{"created": {...}}` elements. The order of the contracts is the same as on the ledger.

3.3.8 Exercise by Contract Key

The JSON command below, demonstrates how to exercise the `Archive` choice on the `Account` contract with a `(Party, Text)` *contract key* defined like this:

```
template Account with
    owner : Party
    number : Text
    status : AccountStatus
  where
    signatory owner
    key (owner, number) : (Party, Text)
    maintainer key._1
```

3.3.8.1 HTTP Request

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

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

Where:

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

3.3.8.2 HTTP Response

Formatted similar to *Exercise by Contract ID response*.

3.3.9 Create and Exercise in the Same Transaction

This command allows creating a contract and exercising a choice on the newly created contract in the same transaction.

3.3.9.1 HTTP Request

URL: `/v1/create-and-exercise`
Method: `POST`
Content-Type: `application/json`
Content:

```
{
  "templateId": "Iou:Iou",
  "payload": {
    "observers": [],
    "issuer": "Alice",
    "amount": "999.99",
    "currency": "USD",
    "owner": "Alice"
  },
  "choice": "Iou_Transfer",
  "argument": {
    "newOwner": "Bob"
  }
}
```

Where:

`templateId` - the initial contract template identifier, in the same format as in the *create request*,
`payload` - the initial contract fields as defined in the DAML template and formatted according to *DAML-LF JSON Encoding*,
`choice` - DAML contract choice, that is being exercised,
`argument` - contract choice argument(s).

3.3.9.2 HTTP Response

Please note that the response below is for a consuming choice, so it contains:

`created` and `archived` events for the initial contract (`"contractId": "#1:0"`), which was created and archived right away when a consuming choice was exercised on it,
a `created` event for the contract that is the result of exercising the choice (`"contractId": "#1:2"`).
Content-Type: `application/json`
Content:

```
{
  "result": {
```

(continues on next page)

(continued from previous page)

```
    "exerciseResult": "#1:2",
    "events": [
      {
        "created": {
          "observers": [],
          "agreementText": "",
          "payload": {
            "observers": [],
            "issuer": "Alice",
            "amount": "999.99",
            "currency": "USD",
            "owner": "Alice"
          },
          "signatories": [
            "Alice"
          ],
          "contractId": "#1:0",
          "templateId":
↪"a3b788b4dc18dc060bfb82366ae6dc055b1e361d646d5cfdb1b729607e344336:Iou:Iou
↪"
        }
      },
      {
        "archived": {
          "contractId": "#1:0",
          "templateId":
↪"a3b788b4dc18dc060bfb82366ae6dc055b1e361d646d5cfdb1b729607e344336:Iou:Iou
↪"
        }
      },
      {
        "created": {
          "observers": [
            "Bob"
          ],
          "agreementText": "",
          "payload": {
            "iou": {
              "observers": [],
              "issuer": "Alice",
              "amount": "999.99",
              "currency": "USD",
              "owner": "Alice"
            },
            "newOwner": "Bob"
          },
          "signatories": [
            "Alice"
          ],
```

(continues on next page)

(continued from previous page)

```
          "contractId": "#1:2",
          "templateId":
↪"a3b788b4dc18dc060bfb82366ae6dc055b1e361d646d5cfdb1b729607e344336:Iou:IouTransfer
↪"
        }
      }
    ]
  },
  "status": 200
}
```

3.3.10 Fetch Contract by Contract ID

3.3.10.1 HTTP Request

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

application/json body:

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

3.3.10.2 Contract Not Found HTTP Response

Content-Type: application/json
Content:

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

3.3.10.3 Contract Found HTTP Response

Content-Type: application/json
Content:

```
{
    "status": 200,
    "result": {
        "observers": [],
        "agreementText": "",
        "payload": {
            "iou": {
                "observers": [],
                "issuer": "Alice",
```

(continues on next page)

(continued from previous page)

```
                "amount": "999.99",
                "currency": "USD",
                "owner": "Alice"
            },
            "newOwner": "Alice"
        },
        "signatories": [
            "Alice"
        ],
        "contractId": "#201:1",
        "templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Iou:IouTransfer
↪"
    }
}
```

3.3.11 Fetch Contract by Key

3.3.11.1 HTTP Request

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

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

3.3.11.2 Contract Not Found HTTP Response

Content-Type: application/json
Content:

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

3.3.11.3 Contract Found HTTP Response

Content-Type: application/json
Content:

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

(continues on next page)

(continued from previous page)

```
    "result": {
        "observers": [],
        "agreementText": "",
        "payload": {
            "owner": "Alice",
            "number": "abc123",
            "status": {
                "tag": "Enabled",
                "value": "2020-01-01T00:00:01Z"
            }
        },
        "signatories": [
            "Alice"
        ],
        "key": {
            "_1": "Alice",
            "_2": "abc123"
        },
        "contractId": "#697:0",
        "templateId":
↪"11c8f3ace75868d28136adc5cfc1de265a9ee5ad73fe8f2db97510e3631096a2:Account:Account
↪"
    }
}
```

3.3.12 Get all Active Contracts

List all currently active contracts for all known templates.

Note: Retrieved contracts do not get persisted into a query store database. Query store is a search index and can be used to optimize search latency. See *Start HTTP service* for information on how to start JSON API service with a query store enabled.

Note: You can only query active contracts with the `/v1/query` endpoint. Archived contracts (those that were archived or consumed during an exercise operation) will not be shown in the results.

3.3.12.1 HTTP Request

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

3.3.12.2 HTTP Response

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

3.3.13 Get all Active Contracts Matching a Given Query

List currently active contracts that match a given query.

3.3.13.1 HTTP Request

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

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

Where:

`templateIds` – an array of contract template identifiers to search through,
`query` – search criteria to apply to the specified `templateIds`, formatted according to the *Query language*.

3.3.13.2 Empty HTTP Response

Content-Type: `application/json`
Content:

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

3.3.13.3 Nonempty HTTP Response

Content-Type: `application/json`
Content:

```
{
    "result": [
        {
            "observers": [],
            "agreementText": "",
            "payload": {
                "observers": [],
                "issuer": "Alice",
                "amount": "999.99",
                "currency": "USD",
                "owner": "Alice"
            },
            "signatories": [
                "Alice"
            ],
            "contractId": "#52:0",
```

(continues on next page)

(continued from previous page)

```
            "templateId":
↪"b10d22d6c2f2fae41b353315cf893ed66996ecb0abe4424ea6a81576918f658a:Iou:Iou
↪"
        }
    ],
    "status": 200
}
```

Where

result contains an array of contracts, each contract formatted according to *DAML-LF JSON Encoding*,
status matches the HTTP status code returned in the HTTP header.

3.3.13.4 Nonempty HTTP Response with Unknown Template IDs Warning

Content-Type: application/json
Content:

```
{
    "warnings": {
        "unknownTemplateIds": ["UnknownModule:UnknownEntity"]
    },
    "result": [
        {
            "observers": [],
            "agreementText": "",
            "payload": {
                "observers": [],
                "issuer": "Alice",
                "amount": "999.99",
                "currency": "USD",
                "owner": "Alice"
            },
            "signatories": [
                "Alice"
            ],
            "contractId": "#52:0",
            "templateId":
↪"b10d22d6c2f2fae41b353315cf893ed66996ecb0abe4424ea6a81576918f658a:Iou:Iou
↪"
        }
    ],
    "status": 200
}
```

3.3.14 Fetch Parties by Identifiers

URL: /v1/parties
Method: POST
Content-Type: application/json

Content:

```
["Alice", "Bob", "Dave"]
```

If an empty JSON array is passed: `[]`, this endpoint returns BadRequest(400) error:

```
{
  "status": 400,
  "errors": [
    "JsonReaderError. Cannot read JSON: <[]>. Cause: spray.json.
↪DeserializationException: must be a list with at least 1 element"
  ]
}
```

3.3.14.1 HTTP Response

Content-Type: `application/json`
Content:

```
{
  "status": 200,
  "result": [
    {
      "identifier": "Alice",
      "displayName": "Alice & Co. LLC",
      "isLocal": true
    },
    {
      "identifier": "Bob",
      "displayName": "Bob & Co. LLC",
      "isLocal": true
    },
    {
      "identifier": "Dave",
      "isLocal": true
    }
  ]
}
```

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

Where

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

3.3.14.2 Response with Unknown Parties Warning

Content-Type: `application/json`
Content:

```
{
  "result": [
    {
      "identifier": "Alice",
      "displayName": "Alice & Co. LLC",
      "isLocal": true
    }
  ],
  "warnings": {
    "unknownParties": ["Erin"]
  },
  "status": 200
}
```

The `result` might be an empty JSON array if none of the requested parties is known.

3.3.15 Fetch All Known Parties

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

3.3.15.1 HTTP Response

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

3.3.16 Allocate a New Party

This endpoint is a JSON API proxy for the Ledger API's *AllocatePartyRequest*. For more information about party management, please refer to Provisioning Identifiers part of the Ledger API documentation.

3.3.16.1 HTTP Request

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

```
{
  "identifierHint": "Carol",
  "displayName": "Carol & Co. LLC"
}
```

Please refer to *AllocateParty* documentation for information about the meaning of the fields.

All fields in the request are optional, this means that an empty JSON object is a valid request to allocate a new party:

```
{}
```

3.3.16.2 HTTP Response

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

3.3.17 List All DALF Packages

3.3.17.1 HTTP Request

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

3.3.17.2 HTTP Response

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

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

3.3.18 Download a DALF Package

3.3.18.1 HTTP Request

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

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

3.3.18.2 HTTP Response, status: 200 OK

Transfer-Encoding: `chunked`
Content-Type: `application/octet-stream`
Content: <DALF bytes>

The content (body) of the HTTP response contains raw DALF package bytes, without any encoding. Note that the package ID specified in the URL is actually the SHA-256 hash of the downloaded DALF package and can be used to validate the integrity of the downloaded content.

3.3.18.3 HTTP Response with Error, any status different from 200 OK

Any status different from `200 OK` will be in the format specified below.

Content-Type: `application/json`
Content:

```
{
    "errors": [
        "io.grpc.StatusRuntimeException: NOT_FOUND"
    ],
    "status": 500
}
```

3.3.19 Upload a DAR File

3.3.19.1 HTTP Request

URL: `/v1/packages`
Method: `POST`
Content-Type: `application/octet-stream`
Content: <DAR bytes>

The content (body) of the HTTP request contains raw DAR file bytes, without any encoding.

3.3.19.2 HTTP Response, status: 200 OK

Content-Type: `application/json`
Content:

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

3.3.19.3 HTTP Response with Error

Content-Type: `application/json`
Content:

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

3.3.20 Streaming API

Two subprotocols must be passed with every request, as described in *Passing token with WebSockets*.

JavaScript/Node.js example demonstrating how to establish Streaming API connection:

```
const wsProtocol = "daml.ws.auth";
const tokenPrefix = "jwt.token.";
const jwt =
  "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJodHRwczovL2RhbWwuY29tL2xlZGdlci1hcGkiOnsibGVkZ2VySWQiOiJNeUxlZGdlciIsImFwcGxpY
↪34zzF_fbWv7p60r5s1kKzwndvGdsJDX-W4Xhm4oVdp";
const subprotocols = [`${tokenPrefix}${jwt}`, wsProtocol];

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

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

ws.addEventListener("message", function incoming(data) {
  console.log(data);
});
```

Please note that Streaming API does not allow multiple requests over the same WebSocket connection. The server returns an error and disconnects if second request received over the same WebSocket connection.

3.3.20.1 Error and Warning Reporting

Errors and warnings reported as part of the regular `on-message` flow: `ws.addEventListener("message", ...)`.

Streaming API error messages formatted the same way as *synchronous API errors*.

Streaming API reports only one type of warnings – unknown template IDs, which is formatted as:

```
{"warnings":{"unknownTemplateIds":<JSON Array of template ID strings>>}}
```

Error and Warning Examples

```
{"warnings": {"unknownTemplateIds": ["UnknownModule:UnknownEntity"]}}

{
  "errors":["JsonReaderError. Cannot read JSON: <{\"templateIds\":[]}>.
↪Cause: spray.json.DeserializationException: search requires at least one
↪item in 'templateIds'"],
  "status":400
}

{
  "errors":["Multiple requests over the same WebSocket connection are not
↪allowed."],
  "status":400
}
```

(continues on next page)

(continued from previous page)

```
{
  "errors":["Could not resolve any template ID from request."],
  "status":400
}
```

3.3.20.2 Contracts Query Stream

URL: `/v1/stream/query`
Scheme: `ws`
Protocol: `WebSocket`

List currently active contracts that match a given query, with continuous updates.

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

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

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

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

An optional `offset` returned by a prior query (see output examples below) may be specified *before* the above, as a separate body. It must be a string, and if specified, the stream will begin immediately *after* the response body that included that offset:

```
{"offset": "5609"}
```

The output is a series of JSON documents, each `payload` formatted according to *DAML-LF JSON Encoding*:

```
{
    "events": [{
        "created": {
            "observers": [],
            "agreementText": "",
            "payload": {
                "observers": [],
                "issuer": "Alice",
                "amount": "999.99",
                "currency": "USD",
                "owner": "Alice"
            },
            "signatories": ["Alice"],
            "contractId": "#1:0",
            "templateId":
↪"eb3b150383a979d6765b8570a17dd24ae8d8b63418ee5fd20df20ad2a1c13976:Iou:Iou
↪"
```

(continues on next page)

(continued from previous page)

```
        },
        "matchedQueries": [1, 2]
    }]
}
```

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

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

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

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

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

where offset is the last seen ledger offset.

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

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

(continues on next page)

(continued from previous page)

```
            "templateId":
↪"eb3b150383a979d6765b8570a17dd24ae8d8b63418ee5fd20df20ad2a1c13976:Iou:Iou
↪"
        },
        "matchedQueries": [0, 2]
    }, {
        "created": {
            "observers": [],
            "agreementText": "",
            "payload": {
                "observers": [],
                "issuer": "Alice",
                "amount": "957.57",
                "currency": "USD",
                "owner": "Alice"
            },
            "signatories": ["Alice"],
            "contractId": "#2:2",
            "templateId":
↪"eb3b150383a979d6765b8570a17dd24ae8d8b63418ee5fd20df20ad2a1c13976:Iou:Iou
↪"
        },
        "matchedQueries": [1, 2]
    }],
    "offset": "3"
}
```

If any template IDs are found not to resolve, the first element of the stream will report them:

```
{"warnings": {"unknownTemplateIds": ["UnknownModule:UnknownEntity"]}}
```

and the stream will continue, provided that at least one template ID resolved properly.

Aside from "created" and "archived" elements, "error" elements may appear, which contain a string describing the error. The stream will continue in these cases, rather than terminating.

Some notes on behavior:

1. Each result array means this is what would have changed if you just polled /v1/query iteratively. In particular, just as polling search can miss contracts (as a create and archive can be paired between polls), such contracts may or may not appear in any result object.
2. No archived ever contains a contract ID occurring within a created in the same array. So, for example, supposing you are keeping an internal map of active contracts keyed by contract ID, you can apply the created first or the archived first, forwards, backwards, or in random order, and be guaranteed to get the same results.
3. Within a given array, if an archived and created refer to contracts with the same template ID and *contract key*, the archived is guaranteed to occur before the created.
4. Except in cases of #3, within a single response array, the order of created and archived is undefined and does not imply that any element occurred before or after any other one.
5. You will almost certainly receive contract IDs in archived that you never received a created for. These are contracts that query filtered out, but for which the server no longer is aware of that. You can safely ignore these. However, such phantom archives *are* guaranteed to

represent an actual archival *on the ledger*, so if you are keeping a more global dataset outside the context of this specific search, you can use that archival information as you wish.

3.3.20.3 Fetch by Key Contracts Stream

URL: `/v1/stream/fetch`
Scheme: `ws`
Protocol: `WebSocket`

List currently active contracts that match one of the given `{templateId, key}` pairs, with continuous updates.

`application/json` body must be sent first, formatted according to the following rule:

```
[
    {"templateId": "<template ID 1>", "key": <key 1>},
    {"templateId": "<template ID 2>", "key": <key 2>},
    ...
    {"templateId": "<template ID N>", "key": <key N>}
]
```

Where:

`templateId` – contract template identifier, same as in *create request*,
`key` – contract key, formatted according to the *DAML-LF JSON Encoding*,

Example:

```
[
    {"templateId": "Account:Account", "key": {"_1": "Alice", "_2": "abc123
↪"}},
    {"templateId": "Account:Account", "key": {"_1": "Alice", "_2": "def345
↪"}}
]
```

The output stream has the same format as the output from the *Contracts Query Stream*. We further guarantee that for every `archived` event appearing on the stream there has been a matching `created` event earlier in the stream, except in the case of missing `contractIdAtOffset` fields in the case described below.

You may supply an optional `offset` for the stream, exactly as with query streams. However, you should supply with each `{templateId, key}` pair a `contractIdAtOffset`, which is the contract ID currently associated with that pair at the point of the given offset, or `null` if no contract ID was associated with the pair at that offset. For example, with the above keys, if you had one "`abc123`" contract but no "`def345`" contract, you might specify:

```
[
    {"templateId": "Account:Account", "key": {"_1": "Alice", "_2": "abc123
↪"},
     "contractIdAtOffset": "#1:0"},
    {"templateId": "Account:Account", "key": {"_1": "Alice", "_2": "def345
↪"},
     "contractIdAtOffset": null}
]
```

If every `contractIdAtOffset` is specified, as is so in the example above, you will not receive any `archived` events for contracts created before the offset *unless* those contracts are identified in a `contractIdAtOffset`. By contrast, if any `contractIdAtOffset` is missing, `archived` event filtering will be disabled, and you will receive phantom archives as with query streams.

3.4 DAML Script

3.4.1 DAML Script Library

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

3.4.1.1 Module Daml.Script

Data Types

data *Commands* a

This is used to build up the commands send as part of `submit`. If you enable the `ApplicativeDo` extension by adding `{-# LANGUAGE ApplicativeDo #-}` at the top of your file, you can use `do`-notation but the individual commands must not depend on each other.

instance Functor *Commands*

instance HasSubmit *Script Commands*

instance Applicative *Commands*

instance HasField commands (SubmitCmd a) (*Commands* a)

instance HasField commands (SubmitMustFailCmd a) (*Commands* a)

data *ParticipantName*

ParticipantName

Field	Type	Description
participantName	Text	

instance HasField participantName *ParticipantName* Text

data *PartyDetails*

The party details returned by the party management service.

PartyDetails

Field	Type	Description
party	Party	Party id
displayName	Optional Text	Optional display name
isLocal	Bool	True if party is hosted by the backing participant.

instance Eq *PartyDetails*

instance Ord *PartyDetails*

instance Show *PartyDetails*

instance HasField continue (ListKnownPartiesPayload a) ([*PartyDetails*] -> a)

instance HasField displayName *PartyDetails* (Optional Text)

instance HasField isLocal *PartyDetails* Bool

instance HasField party *PartyDetails* Party

data *PartyIdHint*

A hint to the backing participant what party id to allocate. Must be a valid PartyIdString (as described in @value.proto@).

PartyIdHint

Field	Type	Description
partyIdHint	Text	

instance HasField partyIdHint *PartyIdHint* Text

data *Script* a

This is the type of A DAML script. `Script` is an instance of `Action`, so you can use `do` notation.

instance Functor *Script*

instance CanAbort *Script*

instance HasSubmit *Script Commands*

instance HasTime *Script*

instance Action *Script*

instance ActionFail *Script*

instance Applicative *Script*

instance HasField runScript (*Script* a) (() -> Free ScriptF (a, ()))

Functions

query : Template t => Party -> *Script* [(ContractId t, t)]
Query the set of active contracts of the template that are visible to the given party.

queryContractId : Template t => Party -> ContractId t -> *Script* (Optional t)
Query for the contract with the given contract id.
Returns `None` if there is no active contract the party is a stakeholder on. This is semantically equivalent to calling `query` and filtering on the client side.

setTime : Time -> *Script* ()
Set the time via the time service.
This is only supported in static time mode when running over the gRPC API.

Note that the ledger time service does not support going backwards in time. However, you can go back in time in DAML Studio.

passTime : RelTime -> *Script* ()

Advance ledger time by the given interval.

Only supported in static time mode when running over the gRPC API and in DAML Studio. Note that this is not an atomic operation over the gRPC API so no other clients should try to change time while this is running.

Note that the ledger time service does not support going backwards in time. However, you can go back in time in DAML Studio.

allocateParty : Text -> *Script* Party

Allocate a party with the given display name using the party management service.

allocatePartyWithHint : Text -> *PartyIdHint* -> *Script* Party

Allocate a party with the given display name and id hint using the party management service.

allocatePartyOn : Text -> *ParticipantName* -> *Script* Party

Allocate a party with the given display name on the specified participant using the party management service.

allocatePartyWithHintOn : Text -> *PartyIdHint* -> *ParticipantName* -> *Script* Party

Allocate a party with the given display name and id hint on the specified participant using the party management service.

listKnownParties : *Script* [*PartyDetails*]

List the parties known to the default participant.

listKnownPartiesOn : *ParticipantName* -> *Script* [*PartyDetails*]

List the parties known to the given participant.

sleep : RelTime -> *Script* ()

Sleep for the given duration.

This is primarily useful in tests where you repeatedly call `query` until a certain state is reached.

Note that this will sleep for the same duration in both wallcock and static time mode.

createCmd : Template t => t -> *Commands* (ContractId t)

Create a contract of the given template.

exerciseCmd : Choice t c r => ContractId t -> c -> *Commands* r

Exercise a choice on the given contract.

exerciseByKeyCmd : (TemplateKey t k, Choice t c r) => k -> c -> *Commands* r

Exercise a choice on the contract with the given key.

createAndExerciseCmd : Choice t c r => t -> c -> *Commands* r

Create a contract and exercise a choice on it in the same transacton.

archiveCmd : Choice t Archive () => ContractId t -> *Commands* ()

Archive the given contract.

`archiveCmd cid` is equivalent to `exerciseCmd cid Archive`.

script : *Script* a -> *Script* a

Convenience helper to declare you are writing a Script.

This is only useful for readability and to improve type inference. Any expression of type `Script a` is a valid script regardless of whether it is implemented using `script` or not.

DAML scenarios provide a simple way for testing DAML models and getting quick feedback in DAML studio. However, scenarios are run in a special process and do not interact with an actual ledger.

This means that you cannot use scenarios to test other ledger clients, e.g., your UI or *DAML triggers*.

DAML Script addresses this problem by providing you with an API with the simplicity of DAML scenarios and all the benefits such as being able to reuse your DAML types and logic while running against an actual ledger in addition to allowing you to experiment in *DAML Studio*. This means that you can use it to test automation logic, your UI but also for *ledger initialization* where scenarios cannot be used (with the exception of *DAML Sandbox*).

You can also use DAML Script interactively using *DAML REPL*.

3.4.2 Usage

Our example for this tutorial consists of 2 templates.

First, we have a template called `Coin`:

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

This template represents a coin issued to `owner` by `issuer`. `Coin` has both the `owner` and the `issuer` as signatories.

Second, we have a template called `CoinProposal`:

```
template CoinProposal
  with
    coin : Coin
  where
    signatory coin.issuer
    observer coin.owner

    choice Accept : ContractId Coin
      controller coin.owner
      do create coin
```

`CoinProposal` is only signed by the `issuer` and it provides a single `Accept` choice which, when exercised by the controller will create the corresponding `Coin`.

Having defined the templates, we can now move on to write DAML scripts that operate on these templates. To get access to the API used to implement DAML scripts, you need to add the `daml-script` library to the `dependencies` field in `daml.yaml`.

```
dependencies:
  - daml-prim
  - daml-stdlib
  - daml-script
```

We also enable the `ApplicativeDo` extension. We will see below why this is useful.

```
{-# LANGUAGE ApplicativeDo #-}

module ScriptExample where

import Daml.Script
```

Since on an actual ledger parties cannot be arbitrary strings, we define a record containing all the parties that we will use in our script so that we can easily swap them out.

```
data LedgerParties = LedgerParties with
  bank : Party
  alice : Party
  bob : Party
```

Let us now write a function to initialize the ledger with 3 `CoinProposal` contracts and accept 2 of them. This function takes the `LedgerParties` as an argument and return something of type `Script ()` which is DAML script's equivalent of `Scenario ()`.

```
initialize : LedgerParties -> Script ()
initialize parties = do
```

First we create the proposals. To do so, we use the `submit` function to submit a transaction. The first argument is the party submitting the transaction. In our case, we want all proposals to be created by the bank so we use `parties.bank`. The second argument must be of type `Commands a` so in our case `Commands (ContractId CoinProposal, ContractId CoinProposal, ContractId CoinProposal)` corresponding to the 3 proposals that we create. `Commands` is similar to `Update` which is used in the `submit` function in scenarios. However, `Commands` requires that the individual commands do not depend on each other. This matches the restriction on the Ledger API where a transaction consists of a list of commands. Using `ApplicativeDo` we can still use `do`-notation as long as we respect this. In `Commands` we use `createCmd` instead of `create` and `exerciseCmd` instead of `exercise`.

```
  (coinProposalAlice, coinProposalBob, coinProposalBank) <- submit parties.
↪bank $ do
    coinProposalAlice <- createCmd (CoinProposal (Coin parties.bank□
↪parties.alice))
    coinProposalBob <- createCmd (CoinProposal (Coin parties.bank parties.
↪bob))
    coinProposalBank <- createCmd (CoinProposal (Coin parties.bank parties.
↪bank))
    pure (coinProposalAlice, coinProposalBob, coinProposalBank)
```

Now that we have created the `CoinProposals`, we want `Alice` and `Bob` to accept the proposal while the `Bank` will ignore the proposal that it has created for itself. To do so we use separate `submit` statements for `Alice` and `Bob` and call `exerciseCmd`.

```
  coinAlice <- submit parties.alice $ exerciseCmd coinProposalAlice Accept
  coinBob <- submit parties.bob $ exerciseCmd coinProposalBob Accept
```

Finally, we call `pure ()` on the last line of our script to match the type `Script ()`.

```
  pure ()
```

We have now defined a way to initialize the ledger so we can write a test that checks that the contracts that we expect exist afterwards.

First, we define the signature of our test. We will create the parties used here in the test, so it does not take any arguments.

```
test : Script ()
test = do
```

Now, we create the parties using the allocateParty function. This uses the party management service to create new parties with the given display name. Note that the display name does not identify a party uniquely. If you call allocateParty twice with the same display name, it will create 2 different parties. This is very convenient for testing since a new party cannot see any old contracts on the ledger so using new parties for each test removes the need to reset the ledger.

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

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

```
  initialize parties
```

To verify the contracts on the ledger, we use the query function. We pass it the type of the template and a party. It will then give us all active contracts of the given type visible to the party. In our example, we expect to see one active CoinProposal for bank and one Coin contract for each of Alice and Bob. We get back list of (ContractId t, t) pairs from query. In our tests, we do not need the contract ids, so we throw them away using map snd.

```
  proposals <- query @CoinProposal bank
  assertEq [CoinProposal (Coin bank bank)] (map snd proposals)

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

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

To run our script, we first build it with daml build and then run it by pointing to the DAR, the name of our script, the host and port our ledger is running on and the time mode of the ledger.

```
daml script --dar .daml/dist/script-example-0.0.1.dar --script-name
ScriptExample:test --ledger-host localhost --ledger-port 6865
```

Up to now, we have worked with parties that we have allocated in the test. We can also pass in the path to a file containing the input in the *DAML-LF JSON Encoding*.

```
{
  "alice": "Alice",
```

(continues on next page)

(continued from previous page)

```
  "bob": "Bob",
  "bank": "Bank"
}
```

We can then initialize our ledger passing in the json file via `--input-file`.

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

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

While we will not use it here, there is also an `--output-file` option that you can use to write the result of a script to a file using the DAML-LF JSON encoding. This is particularly useful if you need to consume the result from another program.

3.4.3 Using DAML Script for Ledger Initialization

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

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

3.4.3.1 Migrating from Scenarios

Existing scenarios that you used for ledger initialization can be translated to DAML script but there are a few things to keep in mind:

1. You need to add `daml-script` to the list of dependencies in your `daml.yaml`.
2. You need to import the `Daml.Script` module.
3. Calls to `create`, `exercise`, `exerciseByKey` and `createAndExercise` need to be suffixed with `Cmd`, e.g., `createCmd`.
4. Instead of specifying a `scenario` field in your `daml.yaml`, you need to specify an `init-script` field. The initialization script is specified via `Module:identifier` for both fields.
5. DAML script only supports the commands available on the ledger API so you cannot call functions like `fetch` directly. This is intentional. Your initialization scripts should not be able to create transactions that a ledger client would not be able to create. If you want to call methods not exposed via the Ledger API, you can create a new template with a single choice and call that via `createAndExercise`.
6. You need to replace calls to `getParty x` by `allocatePartyWithHint x (PartyIdHint x)`.

3.4.4 Using DAML Script in Distributed Topologies

So far, we have run DAML script against a single participant node. It is also more possible to run it in a setting where different parties are hosted on different participant nodes. To do so, pass the `--participant-config participants.json` file to `daml script` instead of `--ledger-host` and `ledger-port`. The file should be of the format

```
{
    "default_participant": {"host": "localhost", "port": 6866, "access_
↪token": "default_jwt", "application_id": "myapp"},
    "participants": {
        "one": {"host": "localhost", "port": 6865, "access_token": "jwt_
↪for_alice", "application_id": "myapp"},
        "two": {"host": "localhost", "port": 6865, "access_token": "jwt_
↪for_bob", "application_id": "myapp"}
    },
    "party_participants": {"alice": "one", "bob": "two"}
}
```

This will define a participant called `one`, a default participant and it defines that the party `alice` is on participant `one`. Whenever you submit something as party, we will use the participant for that party or if none is specified `default_participant`. If `default_participant` is not specified, using a party with an unspecified participant is an error.

`allocateParty` will also use the `default_participant`. If you want to allocate a party on a specific participant, you can use `allocatePartyOn` which accepts the participant name as an extra argument.

3.4.5 Running DAML Script against Ledgers with Authorization

To run DAML Script against a ledger that verifies authorization, you need to specify an access token. There are two ways of doing that:

1. Specify a single access token via `--access-token-file path/to/jwt`. This token will then be used for all requests so it must provide claims for all parties that you use in your script.
2. If you need multiple tokens, e.g., because you only have single-party tokens you can use the `access_token` field in the participant config specified via `--participant-config`. The section on *using DAML Script in distributed topologies* contains an example. Note that you can specify the same participant twice if you want different auth tokens.

If you specify both `--access-token-file` and `--participant-config`, the participant config takes precedence and the token from the file will be used for any participant that does not have a token specified in the config.

3.4.6 Running DAML Script against the HTTP JSON API

In some cases, you only have access to the *HTTP JSON API* but not to the gRPC of a ledger, e.g., on project:DABL. For this usecase, DAML script can be run against the JSON API. Note that if you do have access to the gRPC API, running DAML script against the JSON API does not have any advantages.

To run DAML script against the JSON API you have to pass the `--json-api` parameter to `daml script`. There are a few differences and limitations compared to running DAML Script against the gRPC API:

1. When running against the JSON API, the `--host` argument has to contain an `http://` or `https://` prefix, e.g., `daml script --host http://localhost --port 7575 --json-api`.
2. The JSON API only supports single-command submissions. This means that within a single call to `submit` you can only execute one ledger API command, e.g., one `createCmd` or one `exerciseCmd`.
3. The JSON API requires authorization tokens even when it is run against a ledger that doesn't verify authorization. The section on *authorization* describes how to specify the tokens.
4. The tokens must contain exactly one party in `actAs` and/or `readAs`. This party will be used for `submit` and `query`. Passing a party as the argument to `submit` and `query` that is different from the party in the token is an error.
5. If you use multiple parties within your DAML Script, you need to specify one token per party.
6. `getTime` will always return the Unix epoch in static time mode since the time service is not exposed via the JSON API.
7. `setTime` is not supported and will throw a runtime error.

3.5 DAML REPL

The DAML REPL allows you to use the *DAML Script* API interactively. This is useful for debugging and for interactively inspecting and manipulating a ledger.

3.5.1 Usage

First create a new project based on the `script-example` template. Take a look at the documentation for *DAML Script* for details on this template.

```
daml new script-example --template script-example # create a project
↪called script-example based on the template
cd script-example # switch to the new project
```

Now, build the project and start *DAML Sandbox*, the in-memory ledger included in the DAML SDK. Note that we are starting Sandbox in wallclock mode. Static time is not supported in `daml repl`.

```
daml build
daml sandbox --wall-clock-time --port=6865 .daml/dist/script-example-0.0.1.
↪dar
```

Now that the ledger has been started, you can launch the REPL in a separate terminal using the following command.

```
daml repl --ledger-host=localhost --ledger-port=6865 .daml/dist/script-
↪example-0.0.1.dar --import script-example
```

The `--ledger-host` and `--ledger-port` parameters point to the host and port your ledger is running on. In addition to that, you also need to pass in the name of a DAR containing the templates and other definitions that will be accessible in the REPL. We also specify that we want to import all modules from the `script-example` package. If your modules provide colliding definitions you can also import modules individually from within the REPL. Note that you can also specify multiple DARs and they will all be available.

You should now see a prompt looking like

```
daml>
```

You can think of this prompt like a line in a `do`-block of the `Script` action. Each line of input has to have one of the following two forms:

1. An expression `expr` of type `Script a` for some type `a`. This will execute the script and print the result if `a` is an instance of `Show` and not `()`.
2. A pure expression `expr` of type `a` for some type `a` where `a` is an instance of `Show`. This will evaluate `expr` and print the result. If you are only interest in pure expressions you can also use DAML REPL *without connecting to a ledger*.
3. A binding of the form `pat <- expr` where `pat` is pattern, e.g., a variable name `x` to bind the result to and `expr` is an expression of type `Script a`. This will execute the script and match the result against the pattern `pat` bindings the matches to the variables in the pattern. You can then use those variables on subsequent lines.
4. A `let` binding of the form `let pat = y`, where `pat` is a pattern and `y` is a pure expression or `let f x = y` to define a function. The bound variables can be used on subsequent lines.
5. Next to DAML code the REPL also understands REPL commands which are prefixed by `:`. Enter `:help` to see a list of supported REPL commands.

First create two parties: A party with the display name `"Alice"` and the party id `"alice"` and a party with the display name `"Bob"` and the party id `"bob"`.

```
daml> alice <- allocatePartyWithHint "Alice" (PartyIdHint "alice")
daml> bob <- allocatePartyWithHint "Bob" (PartyIdHint "bob")
```

Next, create a `CoinProposal` from `Alice` to `Bob`

```
daml> submit alice (createCmd (CoinProposal (Coin alice bob)))
```

As Bob, you can now get the list of active `CoinProposal` contracts using the `query` function. The `debug : Show a => a -> Script ()` function can be used to print values.

```
daml> proposals <- query @CoinProposal bob
daml> debug proposals
[Daml.Script:39]: [(<contract-id>,CoinProposal {coin = Coin {issuer =
↪'alice', owner = 'bob'}})]
```

Finally, accept all proposals using the `forA` function to iterate over them.

```
daml> forA proposals $ \(contractId, _) -> submit bob (exerciseCmd
↪contractId Accept)
```

Using the `query` function we can now verify that there is one `Coin` and no `CoinProposal`:

```
daml> coins <- query @Coin bob
daml> debug coins
[Daml.Script:39]: [(<contract-id>,Coin {issuer = 'alice', owner = 'bob'})]
daml> proposals <- query @CoinProposal bob
[Daml.Script:39]: []
```

To exit `daml repl` press `Control-D`.

3.5.2 What is in scope at the prompt?

In the prompt, all modules from DALFs specified in `--import` are imported automatically. In addition to that, the `DAML.Script` module is also imported and gives you access to the DAML Script API.

You can use the commands `:module + ModA ModB` … to import additional modules and `:module - ModA ModB` … to remove previously added imports. Modules can also be imported using regular import declarations instead of `module +`. The command `:show imports` lists the currently active imports.

```
daml> import DA.Time
daml> debug (days 1)
```

3.5.3 Using DAML REPL without a Ledger

If you are only interested in pure expressions, e.g., because you want to test how some function behaves you can omit the `--ledger-host` and `-ledger-port` parameters. DAML REPL will work as usual but any attempts to call DAML Script APIs that interact with the ledger, e.g., `submit` will result in the following error:

```
daml> java.lang.RuntimeException: No default participant
```

3.5.4 Connecting via TLS

You can connect to a ledger that requires TLS by passing `--tls`. A custom root certificate used for validating the server certificate can be set via `--cacrt`. Finally, you can also enable client authentication by passing `--pem client.key --crt client.crt`. If `--cacrt` or `--pem` and `--crt` are passed TLS is automatically enabled so `--tls` is redundant.

3.5.5 Connection to a Ledger with Authorization

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

3.5.6 Using DAML REPL to convert to JSON

Using the `:json` command you can encode serializable DAML expressions as JSON. For example using the definitions and imports from above:

```
daml> :json days 1
{"microseconds":86400000000}
daml> :json map snd coins
[{"issuer":"alice","owner":"bob"}]
```

3.6 Upgrading and extending DAML applications

3.6.1 Automating the Upgrade Process

In this section, we are going to automate the upgrade of our coin process using *DAML Script* and *DAML Triggers*. Note that automation for upgrades is specific to an individual application, just like the upgrade models. Nevertheless, we have found that the pattern shown here occurs frequently.

3.6.1.1 Structuring the Upgrade

There are three kinds of actions performed during the upgrade:

1. Alice creates `UpgradeCoinProposal` contracts. We assume here, that Alice wants to upgrade all `Coin` contracts she has issued. Since the `UpgradeCoinProposal` proposal is specific to each owner, Alice has to create one `UpgradeCoinProposal` per owner. There can be potentially many owners but this step only has to be performed once assuming Alice will not issue more `Coin` contracts after this point.
2. Bob and other owners accept the `UpgradeCoinProposal`. To keep this example simple, we assume that there are only coins issued by Alice. Therefore, each owner has to accept at most one proposal.
3. As owners accept upgrade proposals, Alice has to upgrade each coin. This means that she has to execute the upgrade choice once for each coin. Owners will not all accept the upgrade at the same time and some might never accept it. Therefore, this should be a long-running process that upgrades all coins of a given owner as soon as they accept the upgrade.

Given those constraints, we are going to use the following tools for the upgrade:

1. A DAML script that will be executed once by Alice and creates an `UpgradeCoinProposal` contract for each owner.
2. Navigator to accept the `UpgradeCoinProposal` as Bob. While we could also use a DAML script to accept the proposal, this step will often be exposed as part of a web UI so doing it interactively in Navigator resembles that workflow more closely.
3. A long-running DAML trigger that upgrades all `Coin` contracts for which there is a corresponding `UpgradeCoinAgreement`.

3.6.1.2 Implementation of the DAML Script

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

```
initiateUpgrade : Party -> Script ()
initiateUpgrade issuer = do
  coins <- query @Coin issuer
  let myCoins = filter (\(_cid, c) -> c.issuer == issuer) coins
  let owners = dedup $ map (\(_cid, c) -> c.owner) myCoins
  forA_ owners $ \owner -> do
    debug ("Creating upgrade proposal for: " <> show owner)
    submit issuer $ createCmd (UpgradeCoinProposal issuer owner)
```

3.6.1.3 Implementation of the DAML Trigger

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

```
upgradeTrigger : Trigger ()
upgradeTrigger = Trigger with
  initialize = \_acs -> ()
  updateState = \_acs _msg () -> ()
  registeredTemplates = AllInDar
```

(continues on next page)

(continued from previous page)

```
  heartbeat = None
  rule = triggerRule
```

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

```
triggerRule : Party -> ACS -> Time -> Map CommandId [Command] -> () ->
↪TriggerA ()
triggerRule issuer acs _ _ _ = do
  let agreements =
        filter (\(_cid, agreement) -> agreement.issuer == issuer) $
        getContracts @UpgradeCoinAgreement acs
  let allCoins =
        filter (\(_cid, coin) -> coin.issuer == issuer) $
        getContracts @Coin acs
  forA_ agreements $ \(agreementCid, agreement) -> do
    let coinsForOwner = filter (\(_cid, coin) -> coin.owner == agreement.
↪owner) allCoins
    forA_ coinsForOwner $ \(coinCid, _) ->
      emitCommands
        [exerciseCmd agreementCid (Upgrade coinCid)]
        [toAnyContractId coinCid]
```

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

3.6.1.4 Deploying and Executing the Upgrade

Now that we defined our DAML script and our trigger, it is time to use them! If you still have Sandbox running from the previous section, stop it to clear out all data before continuing.

First, we start sandbox passing in the `coin-upgrade` DAR. Since a DAR includes all transitive dependencies, this includes `coin-1.0.0` and `coin-2.0.0`.

```
$ cd example/coin-upgrade
$ daml sandbox .daml/dist/coin-upgrade-1.0.0.dar
```

To simplify the setup here, we use a DAML script to create 3 parties Alice, Bob and Charlie and two `Coin` contracts issues by Alice, one owned by Bob and one owned by Charlie.

```
setup : Script ()
setup = do
  alice <- allocatePartyWithHint "Alice" (PartyIdHint "Alice")
  bob <- allocatePartyWithHint "Bob" (PartyIdHint "Bob")
  charlie <- allocatePartyWithHint "Charlie" (PartyIdHint "Charlie")
  bobProposal <- submit alice $ createCmd (CoinProposal alice bob)
  submit bob $ exerciseCmd bobProposal CoinProposal_Accept
```

(continues on next page)

(continued from previous page)

```
  charlieProposal <- submit alice $ createCmd (CoinProposal alice charlie)
  submit charlie $ exerciseCmd charlieProposal CoinProposal_Accept
  pure ()
```

Run the script as follows:

```
$ cd example/coin-initiate-upgrade
$ daml build
$ daml script --dar=.daml/dist/coin-initiate-upgrade-1.0.0.dar --script-
↪name=InitiateUpgrade:setup --ledger-host=localhost --ledger-port=6865 --
↪wall-clock-time
```

If you now start Navigator from the `coin-initiate-upgrade` directory and log in as Alice, you can see the two `Coin` contracts.

Next, we run the trigger for Alice. The trigger will keep running throughout the rest of this example.

```
$ cd example/coin-upgrade-trigger
$ daml build
$ daml trigger --dar=.daml/dist/coin-upgrade-trigger-1.0.0.dar --trigger-
↪name=UpgradeTrigger:upgradeTrigger --ledger-host=localhost --ledger-
↪port=6865 --ledger-party=Alice --wall-clock-time
```

With the trigger running, we can now run the script to create the `UpgradeCoinProposal` contracts (we could also have done that before starting the trigger). The script takes an argument of type `Party`. We can pass this in via the `--input-file` argument which we will point to a file `party.json` containing `"Alice"`. This allows us to change the party without having to change the code of the script.

```
$ cd example/coin-initiate-upgrade
$ daml build
$ daml script --dar=.daml/dist/coin-initiate-upgrade-1.0.0.dar --script-
↪name=InitiateUpgrade:initiateUpgrade --ledger-host=localhost --ledger-
↪port=6865 --wall-clock-time --input-file=party.json
```

At this point, our trigger is running and the `UpgradeCoinProposal` contracts for Bob and Charlie have been created. What is left to do is to accept the proposals. Our trigger will then automatically pick them up and upgrade the `Coin` contracts.

First, start Navigator and log in as Bob. Click on the `UpgradeCoinProposal` and accept it. If you now go back to the contracts tab, you can see that the `Coin` contract has been archived and instead there is a new `CoinWithAmount` upgrade. Our trigger has successfully upgraded the `Coin`!

Next, log in as Charlie and accept the `UpgradeCoinProposal`. Just like for Bob, you can see that the `Coin` contract has been archived and instead there is a new `CoinWithAmount` contract.

Since we upgraded all `Coin` contracts issued by Alice, we can now stop the trigger and declare the update successful.

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

In applications backed by a centralized database controlled by a single operator, it is possible to upgrade an application in a single step that migrates all existing data to a new data model.

However, in a DAML application running on a distributed ledger, the signatories of a contract have agreed to one specific version of a template. Changing the definition of a template, e.g., by extending it with a new choice without agreement from signatories of contracts of that template would completely break the authorization guarantees provided by DAML.

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

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

3.6.2 Structuring upgrade contracts

Upgrade contracts are specific to the templates that are being upgraded. However, there are common patterns between most of them. We use the example of a simple *Coin* template as an example here. We have some prescience that there will be future versions of *Coin*, and so place the definition of `Coin` in a module named `CoinV1`

```
module CoinV1 where

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

A *Coin* has an issuer and an owner and both are signatories. Our goal is to extend this *Coin* template with a field that represents the number of coins to avoid needing 1000 contracts to represent 1000 coins. (In a real application, you would also want choices for merging and splitting such a *Coin*. For the sake of simplicity, we omit those here.) We use a different name for the new template here. This is not required as templates are identified by the triple *(PackageId, ModuleName, TemplateName)*

```
module CoinV2 where

template CoinWithAmount
  with
    issuer : Party
    owner : Party
    amount : Int
  where
    signatory issuer, owner
```

Next, we need to provide a way for the signatories, issuer and owner, to agree to a contract being upgraded. It would be possible to structure this such that issuer and owner have to agree to an upgrade for each individual *Coin* contract separately. However, since the template definition for all of them is the same, this is usually not necessary for most applications. Instead, we collect agreement from the signatories only once and use that to upgrade all coins. Since there are multiple signatories involved here, we use a *Propose-Accept workflow*. First, we define an *UpgradeCoinProposal* template that will be created by the issuer. This template has an *Accept* choice that the *owner* can exercise which will then create an *UpgradeCoinAgreement*.

```
template UpgradeCoinProposal
  with
    issuer : Party
    owner : Party
  where
    signatory issuer
    observer owner
    key (issuer, owner) : (Party, Party)
    maintainer key._1
    choice Accept : ContractId UpgradeCoinAgreement
      controller owner
      do create UpgradeCoinAgreement with ..
```

Now we can define the *UpgradeCoinAgreement* template. This template has one *nonconsuming* choice that takes the contract ID of a *Coin* contract, archives this *Coin* contract and creates a *CoinWithAmount* contract with the same issuer and owner and the *amount* set to 1.

```
template UpgradeCoinAgreement
  with
    issuer : Party
    owner : Party
  where
    signatory issuer, owner
    key (issuer, owner) : (Party, Party)
    maintainer key._1
    nonconsuming choice Upgrade : ContractId CoinWithAmount
      with
        coinId : ContractId Coin
      controller issuer
      do coin <- fetch coinId
         assert (coin.issuer == issuer)
         assert (coin.owner == owner)
         archive coinId
         create CoinWithAmount with
           issuer = coin.issuer
           owner = coin.owner
           amount = 1
```

3.6.3 Building and deploying coin-1.0.0

Let's see everything in action by first building and deploying `coin-1.0.0`. After this we'll see how to deploy and upgrade to `coin-2.0.0` containing the `CoinWithAmount` template.

First we'll need a sandbox ledger to which we can deploy.

```
$ daml sandbox --port 6865
```

Now we'll setup the project for the original version of our coin. The project contains the DAML for just the `Coin` template, along with a `CoinProposal` template which will allow us to issue some coins in the example below.

Here is the project config.

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

Now we can build and deploy `coin-1.0.0` of our Coin.

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

3.6.4 Create some coin-1.0.0 coins

Let's create some coins!

We'll use the navigator to connect to the ledger, and create two coins issued by Alice, and owned by Bob.

```
$ cd example/coin-1.0.0
$ daml navigator server localhost 6865
```

We point a browser to http://localhost:4000, and follow the steps:

1. **Login as Alice:**
    1. Select Templates tab.
    2. Create a *CoinProposal* with Alice as issuer and Bob as owner.
    3. Create a 2nd proposal in the same way.
2. **Login as Bob:**
    1. Exercise the *CoinProposal_Accept* choice on both proposal contracts.

3.6.5 Building and deploying coin-2.0.0

Now we setup the project for the improved coins containing the *amount* field. This project contains only the `CoinWithAmount` template. The upgrade templates are in a third `coin-upgrade` package. While it would be possible to include the upgrade templates in the same package, this means that the package containing the new `CoinWithAmount` template depends on the previous version. With the approach taken here of keeping the upgrade templates in a separate package, the `coin-1.0.0` package is no longer needed once we have upgraded all coins.

It's worth stressing here that extensions always need to go into separate packages. We cannot just add the new definitions to the original project, rebuild and re-deploy. This is because the cryptographically computed package identifier would change, and would not match the package identifier of the original `Coin` contracts from `coin-1.0.0` which are live on the ledger.

Here is the new project config:

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

Now we can build and deploy `coin-2.0.0` of our Coin.

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

3.6.6 Building and deploying coin-upgrade

Having built and deployed `coin-1.0.0` and `coin-2.0.0` we are now ready to build the upgrade package `coin-upgrade`. The project config references both `coin-1.0.0` and `coin-2.0.0` via the `data-dependencies` field. This allows us to import modules from the respective packages which allows us to reference templates from packages that we already uploaded to the ledger.

When following this example, `path/to/coin-1.0.0.dar` and `path/to/coin-2.0.0.dar` should be replaced by the relative or absolute path to the DAR file created by building the respective projects. Commonly the `coin-1.0.0` and `coin-2.0.0` projects would be sibling directories in the file systems, so this path would be: `../coin-1.0.0/.daml/dist/coin-1.0.0.dar`.

```
name: coin-upgrade
version: 1.0.0
dependencies:
  - daml-prim
  - daml-stdlib
data-dependencies:
  - path/to/coin-1.0.0.dar
  - path/to/coin-2.0.0.dar
```

The DAML for the upgrade contacts imports the modules for both the new and old coin versions.

```
module UpgradeFromCoinV1 where
import CoinV2
import CoinV1
```

Now we can build and deploy `coin-upgrade`. Note that uploading a DAR also uploads its dependencies so if `coin-1.0.0` and `coin-2.0.0` had not already been deployed before, they would be deployed as part of deploying `coin-upgrade`.

```
$ cd example/coin-upgrade
$ daml build
$ daml ledger upload-dar --port 6865
```

3.6.7 Upgrade existing coins from coin-1.0.0 to coin-2.0.0

We start the navigator again.

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

Finally, we point a browser to http://localhost:4000 and can effect the coin upgrades:

1. **Login as Alice**
   1. Select Templates tab.
   2. Create an `UpgradeCoinProposal` with Alice as issuer and Bob as owner.
2. **Login as Bob**
   1. Exercise the `Accept` choice of the upgrade proposal, creating an `UpgradeCoinAgreement`.
3. **Login again as Alice**
   1. Use the `UpgradeCoinAgreement` repeatedly to upgrade any coin for which Alice is issuer and Bob is owner.

3.6.8 Further Steps

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

3.7 The Ledger API

3.7.1 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*.

3.7.1.1 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`.

3.7.1.2 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 deduplication

The command submission service deduplicates submitted commands based on the submitting *party* and *command ID*:

Applications can provide a *deduplication time* for each command. If this parameter is not set, the default maximum deduplication time is used.
A command submission is considered a duplicate submission if the ledger server receives the command within the deduplication time of a previous command with the same command ID from the same submitting party.
Duplicate command submissions will be ignored until either the deduplication time of the original command has elapsed or the original submission was rejected (i.e. the command failed and resulted in a rejected transaction), whichever comes first.

Command deduplication is only *guaranteed* to work if all commands are submitted to the same participant. Ledgers are free to perform additional command deduplication across participants. Consult the respective ledger's manual for more details.

A command submission will return:

- The result of the submission (`Empty` or a gRPC error), if the command was submitted outside of the deduplication time of a previous command with the same command ID on the same participant.
- The status error `ALREADY_EXISTS`, if the command was discarded by the ledger server because it was sent within the deduplication time of a previous command with the same command ID.

If the ledger provides additional command deduplication across participants, the initial command submission might be successful, but ultimately the command can be rejected if the deduplication check fails on the ledger.

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

Command completion service

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

Completions contain the `commandId` of the completed command, and the completion status of the command. This status indicates failure or success, and your application should use it to update what it knows about commands in flight, and implement any application-specific error recovery.

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

Command service

Use the **command service** when you want to submit a command and wait for it to be executed. This service is similar to the command submission service, but also receives completions and waits until it knows whether or not the submitted command has completed. It returns the completion status of the command execution.

You can use either the command or command submission services to submit commands to effect a ledger change. The command service is useful for simple applications, as it handles a basic form of coordination between command submission and completion, correlating submissions with completions, and returning a success or failure status. This allow simple applications to be completely stateless, and alleviates the need for them to track command submissions.

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

3.7.1.3 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.

3.7.1.4 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 the maximum command deduplication time (see *Command Deduplication* for details).

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

3.7.1.5 Testing services

These are only for use for testing with the Sandbox, not for on production ledgers.

Time service

Use the **time service** to obtain the time as known by the ledger server.

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

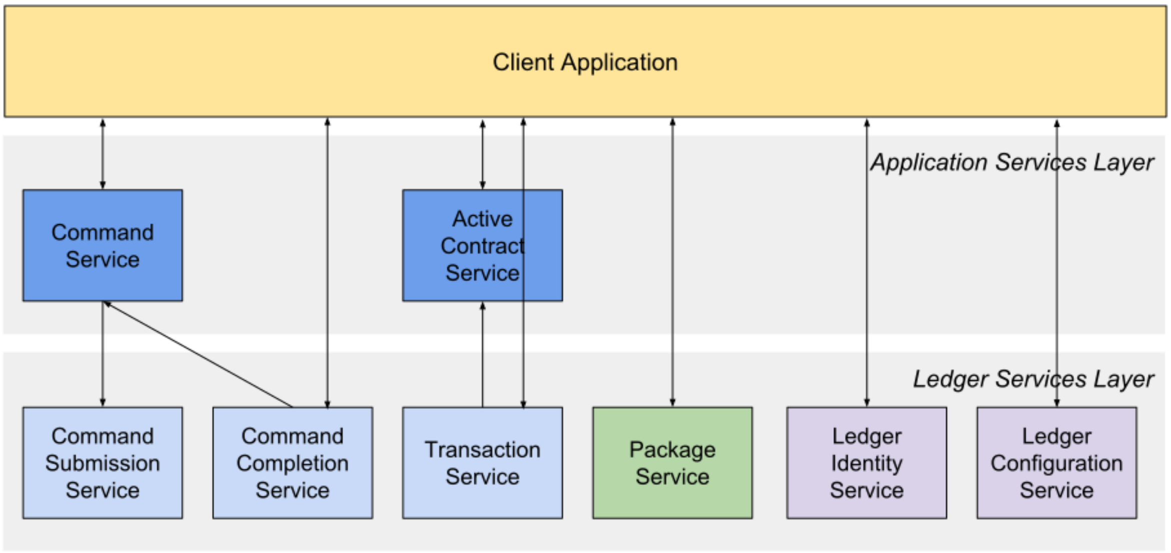
Reset service

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

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

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

3.7.1.6 Services diagram



3.7.2 gRPC

If you want to write an application for the ledger API in other languages, you'll need to use gRPC directly.

If you're not familiar with gRPC and protobuf, we strongly recommend following the gRPC quickstart and gRPC tutorials. This documentation is written assuming you already have an understanding of gRPC.

3.7.2.1 Getting started

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

3.7.2.2 Protobuf reference documentation

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

3.7.2.3 Example project

We have an example project demonstrating the use of the Ledger API with gRPC. To get the example project, `PingPongGrpc`:

1. Configure your machine to use the example by following the instructions at *Set up a Maven project*.
2. Clone the repository from GitHub.
3. Follow the setup instructions in the README. Use `examples.pingpong.grpc.PingPongGrpcMain` as the main class.

About the example project

The example shows very simply how two parties can interact via a ledger, using two DAML contract templates, `Ping` and `Pong`.

The logic of the application goes like this:

1. The application injects a contract of type Ping for Alice.
2. Alice sees this contract and exercises the consuming choice RespondPong to create a contract of type Pong for Bob.
3. Bob sees this contract and exercises the consuming choice RespondPing to create a contract of type Ping for Alice.
4. Points 2 and 3 are repeated until the maximum number of contracts defined in the DAML is reached.

The entry point for the Java code is the main class src/main/java/examples/pingpong/grpc/PingPongGrpcMain.java. Look at it to see how connect to and interact with a ledger using gRPC.

The application prints output like this:

```
Bob is exercising RespondPong on #1:0 in workflow Ping-Alice-1 at count 0
Alice is exercising RespondPing on #344:1 in workflow Ping-Alice-7 at
↪count 9
```

The first line shows:

> Bob is exercising the RespondPong choice on the contract with ID #1:0 for the workflow Ping-Alice-1.
> Count 0 means that this is the first choice after the initial Ping contract.
> The workflow ID Ping-Alice-1 conveys that this is the workflow triggered by the second initial Ping contract that was created by Alice.

This example subscribes to transactions for a single party, as different parties typically live on different participant nodes. However, if you have multiple parties registered on the same node, or are running an application against the Sandbox, you can subscribe to transactions for multiple parties in a single subscription by putting multiple entries into the filters_by_party field of the TransactionFilter message. Subscribing to transactions for an unknown party will result in an error.

3.7.2.4 DAML types and protobuf

For information on how DAML types and contracts are represented by the Ledger API as protobuf messages, see *How DAML types are translated to protobuf*.

3.7.2.5 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. In case of a time constraint violation, please refer to the section *Dealing with time* on how to handle commands with long processing times.

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

OK, INTERNAL, UNKNOWN (when returned by the Command Submission Service) Assume that the command was accepted, and wait for the resulting completion or a timeout from the Command Completion Service.

OK (when returned by the Command Service) You can be sure that the command was successful.

INTERNAL, UNKNOWN (when returned by the Command Service) Resubmit the command with the same command_id.

3.7.3 Ledger API Reference

3.7.3.1 com/daml/ledger/api/v1/active_contracts_service.proto

GetActiveContractsRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as described in `value.proto`). Required
filter	*TransactionFilter*		Templates to include in the served snapshot, per party. Required
verbose	*bool*		If enabled, values served over the API will contain more information than strictly necessary to interpret the data. In particular, setting the verbose flag to true triggers the ledger to include labels for record fields. Optional
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.

3.7.3.2 com/daml/ledger/api/v1/admin/config_management_service.proto

GetTimeModelRequest

GetTimeModelResponse

Field	Type	Label	Description
configuration_generation	*int64*		The current configuration generation. The generation is a monotonically increasing integer that is incremented on each change. Used when setting the time model.
time_model	*TimeModel*		The current ledger time model.

SetTimeModelRequest

Field	Type	Label	Description
submission_id	*string*		Submission identifier used for tracking the request and to reject duplicate submissions. Required.
maximum_record_time	*google.protobuf.Timestamp*		Deadline for the configuration change after which the change is rejected.
configuration_generation	*int64*		The current configuration generation which we're submitting the change against. This is used to perform a compare-and-swap of the configuration to safeguard against concurrent modifications. Required.
new_time_model	*TimeModel*		The new time model that replaces the current one. Required.

SetTimeModelResponse

Field	Type	Label	Description
configuration_generation	*int64*		The configuration generation of the committed time model.

TimeModel

Field	Type	Label	Description
avg_transaction_latency	google.protobuf.Duration		The expected average latency of a transaction, i.e., the average time from submitting the transaction to a [[WriteService]] and the transaction being assigned a record time. Required.
min_skew	google.protobuf.Duration		The minimimum skew between ledger time and record time: lt_TX >= rt_TX - minSkew Required.
max_skew	google.protobuf.Duration		The maximum skew between ledger time and record time: lt_TX <= rt_TX + maxSkew Required.

ConfigManagementService

Ledger configuration management service provides methods for the ledger administrator to change the current ledger configuration. The services provides methods to modify different aspects of the configuration.

Method name	Request type	Response type	Description
GetTimeModel	*GetTimeModelRequest*	*GetTimeModelResponse*	Return the currently active time model and the current configuration generation.
SetTimeModel	*SetTimeModelRequest*	*SetTimeModelResponse*	Set the ledger time model. In case of failure this method responds with: - INVALID_ARGUMENT if arguments are invalid, or the provided configuration generation does not match the current active configuration generation. The caller is expected to retry by again fetching current time model using 'GetTimeModel', applying changes and resubmitting. - ABORTED if the request is rejected or times out. Note that a timed out request may have still been committed to the ledger. Application should re-query the current time model before retrying. - UNIMPLEMENTED if this method is not supported by the backing ledger.

3.7.3.3 com/daml/ledger/api/v1/admin/package_management_service.proto

ListKnownPackagesRequest

ListKnownPackagesResponse

Field	Type	Label	Description
package_details	*PackageDetails*	repeated	The details of all DAML-LF packages known to backing participant. Required

PackageDetails

Field	Type	Label	Description
package_id	*string*		The identity of the DAML-LF package. Must be a valid PackageIdString (as describe in `value.proto`). Required
package_size	*uint64*		Size of the package in bytes. The size of the package is given by the size of the `daml_lf` ArchivePayload. See further details in `daml_lf.proto`. Required
known_since	google.protobuf.Timestamp		Indicates since when the package is known to the backing participant. Required
source_description	*string*		Description provided by the backing participant describing where it got the package from. Optional

UploadDarFileRequest

Field	Type	Label	Description
dar_file	*bytes*		Contains a DAML archive DAR file, which in turn is a jar like zipped container for `daml_lf` archives. See further details in `daml_lf.proto`. Required
submission_id	*string*		Unique submission identifier. Optional, defaults to a random identifier.

UploadDarFileResponse

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

PackageManagementService

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.

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

AllocatePartyRequest

Field	Type	Label	Description
party_id_hint	string		A hint to the backing participant which party ID to allocate. It can be ignored. Must be a valid PartyIdString (as described in `value.proto`). Optional
display_name	string		Human-readable name of the party to be added to the participant. It doesn't have to be unique. Optional

AllocatePartyResponse

Field	Type	Label	Description
party_details	PartyDetails		

GetParticipantIdRequest

GetParticipantIdResponse

Field	Type	Label	Description
participant_id	string		Identifier of the participant, which SHOULD be globally unique. Must be a valid LedgerString (as describe in `value.proto`).

GetPartiesRequest

Field	Type	Label	Description
parties	*string*	repeated	The stable, unique identifier of the DAML parties. Must be valid PartyIdStrings (as described in `value.proto`). Required

GetPartiesResponse

Field	Type	Label	Description
party_details	*PartyDetails*	repeated	The details of the requested DAML parties by the participant, if known. The party details may not be in the same order as requested. Required

ListKnownPartiesRequest

ListKnownPartiesResponse

Field	Type	Label	Description
party_details	*PartyDetails*	repeated	The details of all DAML parties 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
GetParties	*GetPartiesRequest*	*GetPartiesResponse*	Get the party details of the given parties. Only known parties will be returned in the list. This request will always succeed.
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 surface protocol, the arguments take implicit values

3.7.3.5 com/daml/ledger/api/v1/command_completion_service.proto

Checkpoint

Checkpoints may be used to:

detect time out of commands.
provide an offset which can be used to restart consumption.

Field	Type	Label	Description
record_time	google.protobuf.Timestamp		All commands with a maximum record time below this value MUST be considered lost if their completion has not arrived before this checkpoint. Required
offset	*LedgerOffset*		May be used in a subsequent CompletionStreamRequest to resume the consumption of this stream at a later time. Required

CompletionEndRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. 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 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
CompletionStream	*CompletionStreamRequest*	*CompletionStreamResponse*	Subscribe to command completion events.
CompletionEnd	*CompletionEndRequest*	*CompletionEndResponse*	Returns the offset after the latest completion.

3.7.3.6 com/daml/ledger/api/v1/command_service.proto

SubmitAndWaitForTransactionIdResponse

Field	Type	Label	Description
transaction_id	*string*		The id of the transaction that resulted from the submitted command. Must be a valid LedgerString (as described in `value.proto`). Required

SubmitAndWaitForTransactionResponse

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

SubmitAndWaitForTransactionTreeResponse

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

SubmitAndWaitRequest

These commands are atomic, and will become transactions.

Field	Type	Label	Description
commands	*Commands*		The commands to be submitted. Required
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.

3.7.3.7 com/daml/ledger/api/v1/command_submission_service.proto

SubmitRequest

The submitted commands will be processed atomically in a single transaction. Moreover, each `Command` in `commands` will be executed in the order specified by the request.

Field	Type	Label	Description
commands	*Commands*		The commands to be submitted in a single transaction. Required
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.

3.7.3.8 com/daml/ledger/api/v1/commands.proto

Command

A command can either create a new contract or exercise a choice on an existing contract.

Field	Type	Label	Description
create	*CreateCommand*		
exercise	*ExerciseCommand*		
exerciseByKey	*ExerciseByKeyCommand*		
createAndExercise	*CreateAndExerciseCommand*		

Commands

A composite command that groups multiple commands together.

Field	Type	Label	Description
ledger_id	string		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as described in `value.proto`). Required
workflow_id	string		Identifier of the on-ledger workflow that this command is a part of. Must be a valid LedgerString (as described in `value.proto`). Optional
application_id	string		Uniquely identifies the application (or its part) that issued the command. This is used in tracing across different components and to let applications subscribe to their own submissions only. Must be a valid LedgerString (as described in `value.proto`). Required
command_id	string		Uniquely 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
commands	Command	repeated	Individual elements of this atomic command. Must be non-empty. Required
deduplication_time	google.protobuf.Duration		The length of the time window during which all commands with the same party and command ID will be deduplicated. Duplicate commands submitted before the end of this window return an ALREADY_EXISTS error. Optional
min_ledger_time_abs	google.protobuf.Timestamp		Lower bound for the ledger time assigned to the resulting transaction. Note: The ledger time of a transaction is assigned as part of command interpretation. Use this property if you expect that command interpretation will take a considerate amount of time, such that by the time the resulting transaction is sequenced, its assigned ledger time is not valid anymore. Must not be set at the same time as min_ledger_time_rel. Optional
min_ledger_time_rel	google.protobuf.Duration		Same as min_ledger_time_abs, but specified as a duration, starting from the time the command is received by the server. Must not be set at the same time as min_ledger_time_abs. Optional

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

3.7.3.9 com/daml/ledger/api/v1/completion.proto

Completion

A completion represents the status of a submitted command on the ledger: it can be successful or failed.

Field	Type	Label	Description
command_id	*string*		The ID of the succeeded or failed command. Must be a valid LedgerString (as described in `value.proto`). Required
status	google.rpc.Status		Identifies the exact type of the error. For example, malformed or double spend transactions will result in a `INVALID_ARGUMENT` status. Transactions with invalid time time windows (which may be valid at a later date) will result in an `ABORTED` error. Optional
transaction_id	*string*		The transaction_id of the transaction that resulted from the command with command_id. Only set for successfully executed commands. Must be a valid LedgerString (as described in `value.proto`). Optional
trace_context	*TraceContext*		The trace context submitted with the command. This field is a future extension point and is currently not supported. Optional

3.7.3.10 com/daml/ledger/api/v1/event.proto

ArchivedEvent

Records that a contract has been archived, and choices may no longer be exercised on it.

Field	Type	Label	Description
event_id	*string*		The ID of this particular event. Must be a valid LedgerString (as described in `value.proto`). Required
contract_id	*string*		The ID of the archived contract. Must be a valid LedgerString (as described in `value.proto`). Required
template_id	*Identifier*		The template of the archived contract. Required
witness_parties	*string*	repeated	The parties that are notified of this event. For *ArchivedEvent's, these are the intersection of the stakeholders of the contract in question and the parties specified in the 'TransactionFilter.* The stakeholders are the union of the signatories and the observers of the contract. Each one of its elements must be a valid PartyIdString (as 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. When a *CreatedEvent* is returned as part of a transaction tree, this will include all the parties specified in the *TransactionFilter* that are informees of the event. If served as part of a flat transaction those will be limited to all parties specified in the *TransactionFilter* that are stakeholders of the contract (i.e. either signatories or observers). Required
signatories	*string*	repeated	The signatories for this contract as specified by the template. Required
observers	*string*	repeated	The observers for this contract as specified explicitly by the template or implicitly as choice controllers. 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

3.7.3.11 com/daml/ledger/api/v1/ledger_configuration_service.proto

GetLedgerConfigurationRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as described in `value.proto`). Required
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
max_deduplication_time	google.protobuf.Duration		The maximum value for the `deduplication_time` parameter of command submissions (as described in `commands.proto`). This defines the maximum time window during which commands can be deduplicated.

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.

3.7.3.12 com/daml/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.

3.7.3.13 com/daml/ledger/api/v1/ledger_offset.proto

LedgerOffset

Describes a specific point on the ledger.

The Ledger API endpoints that take offsets allow to specify portions of the ledger that are relevant for the client to read.

Offsets returned by the Ledger API can be used as-is (e.g. to keep track of processed transactions and provide a restart point to use in case of need).

The format of absolute offsets is opaque to the client: no client-side transformation of an offset is guaranteed to return a meaningful offset.

The server implementation ensures internally that offsets are lexicographically comparable.

Field	Type	Label	Description
absolute	*string*		The format of this string is specific to the ledger and opaque to the client.
boundary	*LedgerOffset.LedgerBoundary*		

LedgerOffset.LedgerBoundary

Name	Number	Description
LEDGER_BEGIN	0	Refers to the first transaction.
LEDGER_END	1	Refers to the currently last transaction, which is a moving target.

3.7.3.14 com/daml/ledger/api/v1/package_service.proto

GetPackageRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as described in `value.proto`). Required
package_id	*string*		The ID of the requested package. Must be a valid PackageIdString (as described in `value.proto`). Required
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.

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

ResetRequest

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

ResetService

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

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

The semantics are as follows:

> When the reset service returns the reset is initiated, but not completed;

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

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

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

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

3.7.3.16 com/daml/ledger/api/v1/testing/time_service.proto

GetTimeRequest

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

GetTimeResponse

Field	Type	Label	Description
current_time	google.protobuf.Timestamp		The current time according to the ledger server.

SetTimeRequest

Field	Type	Label	Description
ledger_id	*string*		Must correspond to the ledger ID reported by the Ledger Identification Service. Must be a valid LedgerString (as describe in `value.proto`). Required
current_time	google.protobuf.Timestamp		MUST precisely match the current time as it's known to the ledger server. 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.

3.7.3.17 com/daml/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.

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

Transaction

Filtered view of an on-ledger transaction.

Field	Type	Label	Description
transaction_id	*string*		Assigned by the server. Useful for correlating logs. Must be a valid LedgerString (as described in `value.proto`). Required
command_id	*string*		The ID of the command which resulted in this transaction. Missing for everyone except the submitting party. Must be a valid LedgerString (as described in `value.proto`). Optional
workflow_id	*string*		The workflow ID used in command submission. Must be a valid LedgerString (as described in `value.proto`). Optional
effective_at	google.protobuf.Timestamp		Ledger effective time. Must be a valid LedgerString (as described in `value.proto`). Required
events	*Event*	repeated	The collection of events. Only contains `CreatedEvent` or `ArchivedEvent`. Required
offset	*string*		The absolute offset. The format of this field is described in `ledger_offset.proto`. Required
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*		

3.7.3.19 com/daml/ledger/api/v1/transaction_filter.proto

Filters

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

InclusiveFilters

If no internal fields are set, no 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*		

3.7.3.20 com/daml/ledger/api/v1/transaction_service.proto

GetFlatTransactionResponse

Field	Type	Label	Description
transaction	*Transaction*		

GetLedgerEndRequest

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

3.7.3.21 com/daml/ledger/api/v1/value.proto

Enum

A value with finite set of alternative representations.

Field	Type	Label	Description
enum_id	*Identifier*		Omitted from the transaction stream when verbose streaming is not enabled. Optional when submitting commands.
constructor	*string*		Determines which of the Variant's alternatives is encoded in this message. Must be a valid NameString. Required

GenMap

Field	Type	Label	Description
entries	*GenMap.Entry*	repeated	

GenMap.Entry

Field	Type	Label	Description
key	*Value*		
value	*Value*		

Identifier

Unique identifier of an entity.

Field	Type	Label	Description
package_id	*string*		The identifier of the DAML package that contains the entity. Must be a valid PackageIdString. Required
module_name	*string*		The dot-separated module name of the identifier. Required
entity_name	*string*		The dot-separated name of the entity (e.g. record, template,) within the module. Required

List

A homogenous collection of values.

Field	Type	Label	Description
elements	*Value*	repeated	The elements must all be of the same concrete value type. Optional

Map

Field	Type	Label	Description
entries	*Map.Entry*	repeated	

Map.Entry

Field	Type	Label	Description
key	*string*		
value	*Value*		

Optional

Corresponds to Java's Optional type, Scala's Option, and Haskell's Maybe. The reason why we need to wrap this in an additional `message` is that we need to be able to encode the `None` case in the `Value` oneof.

Field	Type	Label	Description
value	*Value*		optional

Record

Contains nested values.

Field	Type	Label	Description
record_id	*Identifier*		Omitted from the transaction stream when verbose streaming is not enabled. Optional when submitting commands.
fields	*RecordField*	repeated	The nested values of the record. Required

RecordField

A named nested value within a record.

Field	Type	Label	Description
label	*string*		When reading a transaction stream, it's omitted if verbose streaming is not enabled. When submitting a commmand, it's optional: - if all keys within a single record are present, the order in which fields appear does not matter. however, each key must appear exactly once. - if any of the keys within a single record are omitted, the order of fields MUST match the order of declaration in the DAML template. Must be a valid NameString
value	*Value*		A nested value of a record. Required

Value

Encodes values that the ledger accepts as command arguments and emits as contract arguments.

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

Field	Type	Label	Description
record	*Record*		
variant	*Variant*		
contract_id	*string*		Identifier of an on-ledger contract. Commands which reference an unknown or already archived contract ID will fail. Must be a valid LedgerString.
list	*List*		Represents a homogeneous list of values.
int64	*sint64*		
numeric	*string*		A Numeric, that is a decimal value with precision 38 (at most 38 significant digits) and a scale between 0 and 37 (significant digits on the right of the decimal point). The field has to match the regex [+-]?d{1,38}(.d{0,37})? and should be representable by a Numeric without loss of precision.
text	*string*		A string.
timestamp	*sfixed64*		Microseconds since the UNIX epoch. Can go backwards. Fixed since the vast majority of values will be greater than 2^28, since currently the number of microseconds since the epoch is greater than that. Range: 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999Z, so that we can convert to/from https://www.ietf.org/rfc/rfc3339.txt
party	*string*		An agent operating on the ledger. Must be a valid PartyIdString.
bool	*bool*		True or false.
unit	google.protobuf.Empty		This value is used for example for choices that don't take any arguments.
date	*int32*		Days since the unix epoch. Can go backwards. Limited from 0001-01-01 to 9999-12-31, also to be compatible with https://www.ietf.org/rfc/rfc3339.txt
optional	*Optional*		The Optional type, None or Some
map	*Map*		The Map type
enum	*Enum*		The Enum type
gen_map	*GenMap*		The GenMap type

Variant

A value with alternative representations.

Field	Type	Label	Description
variant_id	*Identifier*		Omitted from the transaction stream when verbose streaming is not enabled. Optional when submitting commands.
constructor	*string*		Determines which of the Variant's alternatives is encoded in this message. Must be a valid NameString. Required
value	*Value*		The value encoded within the Variant. Required

3.7.3.22 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

3.7.4 How DAML types are translated to protobuf

This page gives an overview and reference on how DAML types and contracts are represented by the Ledger API as protobuf messages, most notably:

- in the stream of transactions from the *TransactionService*
- as payload for *CreateCommand* and *ExerciseCommand* sent to *CommandSubmissionService* and *CommandService*.

The DAML code in the examples below is written in DAML *1.1*.

3.7.4.1 Notation

The notation used on this page for the protobuf messages is the same as you get if you invoke `protoc --decode=Foo < some_payload.bin`. To illustrate the notation, here is a simple definition of the messages `Foo` and `Bar`:

```
message Foo {
  string field_with_primitive_type = 1;
  Bar field_with_message_type = 2;
}

message Bar {
  repeated int64 repeated_field_inside_bar = 1;
}
```

A particular value of `Foo` is then represented by the Ledger API in this way:

```
{ // Foo
  field_with_primitive_type: "some string"
  field_with_message_type { // Bar
    repeated_field_inside_bar: 17
    repeated_field_inside_bar: 42
    repeated_field_inside_bar: 3
  }
}
```

The name of messages is added as a comment after the opening curly brace.

3.7.4.2 Records and primitive types

Records or product types are translated to *Record*. Here's an example DAML record type that contains a field for each primitive type:

```
data MyProductType = MyProductType {
  intField: Int;
  textField: Text;
  decimalField: Decimal;
  boolField: Bool;
  partyField: Party;
  timeField: Time;
  listField: [Int];
  contractIdField: ContractId SomeTemplate
}
```

And here's an example of creating a value of type *MyProductType*:

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

(continues on next page)

(continued from previous page)

```
    }
  }
  fields { // RecordField
    label: "timeField"
    value { // Value
      timestamp: 1526428800000000
    }
  }
  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"
    }
  }
}
```

3.7.4.3 Variants

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

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

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

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

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

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

Listing 1: mySum1

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

Listing 2: mySum2

```
{ // Value
  variant { // Variant
    variant_id { // Identifier
      package_id: "some-hash"
      name: "Types.MySumType"
    }
    constructor: "MyConstructor2"
    value { // Value
      record { // Record
        fields { // RecordField
          label: "sumTextField"
          value { // Value
            text: "it's a sum"
          }
        }
        fields { // RecordField
          label: "sumBoolField"
          value { // Value
            bool: true
          }
        }
      }
    }
  }
}
```

3.7.4.4 Contract templates

Contract templates are represented as records with the same identifier as the template.

This first example template below contains only the signatory party and a simple choice to exercise:

```
data MySimpleTemplateKey =
```

(continues on next page)

(continued from previous page)

```
  MySimpleTemplateKey
    with
      party: Party

template MySimpleTemplate
    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
```

(continues on next page)

(continued from previous page)

```
          party: "Alice"
        }
      }
    }
  witness_parties: "Alice"
}
```

Exercising a choice

A choice is exercised by sending an *ExerciseCommand*. Taking the same contract template again, exercising the choice MyChoice would result in a command similar to the following:

```
{ // ExerciseCommand
  template_id { // Identifier
    package_id: "some-hash"
    name: "Templates.MySimpleTemplate"
  }
  contract_id: "some-contract-id"
  choice: "MyChoice"
  choice_argument { // Value
    record { // Record
      fields { // RecordField
        label: "parameter"
        value { // Value
          int64: 42
        }
      }
    }
  }
}
```

If the template specifies a key, the *ExerciseByKeyCommand* can be used. It works in a similar way as *ExerciseCommand*, but instead of specifying the contract identifier you have to provide its key. The example above could be rewritten as follows:

```
{ // ExerciseByKeyCommand
  template_id { // Identifier
    package_id: "some-hash"
    name: "Templates.MySimpleTemplate"
  }
  contract_key { // Value
    record { // Record
      fields { // RecordField
        label: "party"
        value { // Value
          party: "Alice"
        }
      }
```

(continues on next page)

(continued from previous page)

```
    }
  }
  choice: "MyChoice"
  choice_argument { // Value
    record { // Record
      fields { // RecordField
        label: "parameter"
        value { // Value
          int64: 42
        }
      }
    }
  }
}
```

3.7.5 How DAML types are translated to DAML-LF

This page shows how types in DAML are translated into DAML-LF. It should help you understand and predict the generated client interfaces, which is useful when you're building a DAML-based application that uses the Ledger API or client bindings in other languages.

For an introduction to DAML-LF, see *DAML-LF*.

3.7.5.1 Primitive types

Built-in data types in DAML have straightforward mappings to DAML-LF.

This section only covers the serializable types, as these are what client applications can interact with via the generated DAML-LF. (Serializable types are ones whose values can be written in a text or binary format. So not function types, `Update` and `Scenario` types, as well as any types built up from those.)

Most built-in types have the same name in DAML-LF as in DAML. These are the exact mappings:

DAML primitive type	DAML-LF primitive type
`Int`	`Int64`
`Time`	`Timestamp`
`()`	`Unit`
`[]`	`List`
`Decimal`	`Decimal`
`Text`	`Text`
`Date`	`Date`
`Party`	`Party`
`Optional`	`Optional`
`ContractId`	`ContractId`

Be aware that only the DAML primitive types exported by the Prelude module map to the DAML-LF primitive types above. That means that, if you define your own type named `Party`, it will not translate to the DAML-LF primitive `Party`.

3.7.5.2 Tuple types

DAML tuple type constructors take types `T1, T2, …, TN` to the type `(T1, T2, …, TN)`. These are exposed in the DAML surface language through the Prelude module.

The equivalent DAML-LF type constructors are `daml-prim:DA.Types:TupleN`, for each particular N (where 2 <= N <= 20). This qualified name refers to the package name (`ghc-prim`) and the module name (`GHC.Tuple`).

For example: the DAML pair type `(Int, Text)` is translated to `daml-prim:DA.Types:Tuple2 Int64 Text`.

3.7.5.3 Data types

DAML-LF has three kinds of data declarations:

Record types, which define a collection of data
Variant or **sum** types, which define a number of alternatives
Enum, which defines simplified **sum** types without type parameters nor argument.

Data type declarations in DAML (starting with the `data` keyword) are translated to record, variant or enum types. It's sometimes not obvious what they will be translated to, so this section lists many examples of data types in DAML and their translations in DAML-LF.

Record declarations

This section uses the syntax for DAML *records* with curly braces.

DAML declaration	DAML-LF translation
`data Foo = Foo { foo1: Int; foo2: Text }`	`record Foo □ { foo1: Int64; foo2: Text }`
`data Foo = Bar { bar1: Int; bar2: Text }`	`record Foo □ { bar1: Int64; bar2: Text }`
`data Foo = Foo { foo: Int }`	`record Foo □ { foo: Int64 }`
`data Foo = Bar { foo: Int }`	`record Foo □ { foo: Int64 }`
`data Foo = Foo {}`	`record Foo □ {}`
`data Foo = Bar {}`	`record Foo □ {}`

Variant declarations

DAML declaration	DAML-LF translation
data Foo = Bar Int | Baz Text	variant Foo □ Bar Int64 | Baz Text
data Foo a = Bar a | Baz Text	variant Foo a □ Bar a | Baz Text
data Foo = Bar Unit | Baz Text	variant Foo □ Bar Unit | Baz Text
data Foo = Bar Unit | Baz	variant Foo □ Bar Unit | Baz Unit
data Foo a = Bar | Baz	variant Foo a □ Bar Unit | Baz Unit
data Foo = Foo Int	variant Foo □ Foo Int64
data Foo = Bar Int	variant Foo □ Bar Int64
data Foo = Foo ()	variant Foo □ Foo Unit
data Foo = Bar ()	variant Foo □ Bar Unit
data Foo = Bar { bar: Int } | Baz Text	variant Foo □ Bar Foo.Bar | Baz Text, record Foo.Bar □ { bar: Int64 }
data Foo = Foo { foo: Int } | Baz Text	variant Foo □ Foo Foo.Foo | Baz Text, record Foo.Foo □ { foo: Int64 }
data Foo = Bar { bar1: Int; bar2: Decimal } | Baz Text	variant Foo □ Bar Foo.Bar | Baz Text, record Foo.Bar □ { bar1: Int64; bar2: Decimal }
data Foo = Bar { bar1: Int; bar2: Decimal } | Baz { baz1: Text; baz2: Date }	data Foo □ Bar Foo.Bar | Baz Foo.Baz, record Foo.Bar □ { bar1: Int64; bar2: Decimal }, record Foo.Baz □ { baz1: Text; baz2: Date }

Enum declarations

DAML declaration	DAML-LF declaration
data Foo = Bar | Baz	enum Foo □ Bar | Baz
data Color = Red | Green | Blue	enum Color □ Red | Green | Blue

Banned declarations

There are two gotchas to be aware of: things you might expect to be able to do in DAML that you can't because of DAML-LF.

The first: a single constructor data type must be made unambiguous as to whether it is a record or a variant type. Concretely, the data type declaration `data Foo = Foo` causes a compile-time error, because it is unclear whether it is declaring a record or a variant type.

To fix this, you must make the distinction explicitly. Write `data Foo = Foo {}` to declare a record type with no fields, or `data Foo = Foo ()` for a variant with a single constructor taking unit argument.

The second gotcha is that a constructor in a data type declaration can have at most one unlabelled argument type. This restriction is so that we can provide a straight-forward encoding of DAML-LF types in a variety of client languages.

Banned declaration	Workaround
data Foo = Foo	data Foo = Foo {} to produce record Foo □ {} OR data Foo = Foo () to produce variant Foo □ Foo Unit
data Foo = Bar	data Foo = Bar {} to produce record Foo □ {} OR data Foo = Bar () to produce variant Foo □ Bar Unit
data Foo = Foo Int Text	Name constructor arguments using a record declaration, for example data Foo = Foo { x: Int; y: Text }
data Foo = Bar Int Text	Name constructor arguments using a record declaration, for example data Foo = Bar { x: Int; y: Text }
data Foo = Bar | Baz Int Text	Name arguments to the Baz constructor, for example data Foo = Bar | Baz { x: Int; y: Text }

3.7.5.4 Type synonyms

Type synonyms (starting with the type keyword) are eliminated during conversion to DAML-LF. The body of the type synonym is inlined for all occurrences of the type synonym name.

For example, consider the following DAML type declarations.

```
type Username = Text
data User = User { name: Username }
```

The Username type is eliminated in the DAML-LF translation, as follows:

```
record User □ { name: Text }
```

3.7.5.5 Template types

A *template declaration* in DAML results in one or more data type declarations behind the scenes. These data types, detailed in this section, are not written explicitly in the DAML program but are created by the compiler.

They are translated to DAML-LF using the same rules as for record declarations above.

These declarations are all at the top level of the module in which the template is defined.

Template data types

Every contract template defines a record type for the parameters of the contract. For example, the template declaration:

```
template Iou
  with
    issuer: Party
    owner: Party
    currency: Text
    amount: Decimal
  where
```

results in this record declaration:

```
data Iou = Iou { issuer: Party; owner: Party; currency: Text; amount:□
↪Decimal }
```

This translates to the DAML-LF record declaration:

```
record Iou □ { issuer: Party; owner: Party; currency: Text; amount:□
↪Decimal }
```

Choice data types

Every choice within a contract template results in a record type for the parameters of that choice. For example, let's suppose the earlier `Iou` template has the following choices:

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

      Transfer: ContractId Iou
        with newOwner: Party
        do
          updateOwner newOwner
```

This results in these two record types:

```
data DoNothing = DoNothing {}
data Transfer = Transfer { newOwner: Party }
```

Whether the choice is consuming or nonconsuming is irrelevant to the data type declaration. The data type is a record even if there are no fields.

These translate to the DAML-LF record declarations:

```
record DoNothing □ {}
record Transfer □ { newOwner: Party }
```

3.7.6 Java bindings

3.7.6.1 Generate Java code from DAML

Introduction

When writing applications for the ledger in Java, you want to work with a representation of DAML templates and data types in Java that closely resemble the original DAML code while still being as true to the native types in Java as possible. To achieve this, you can use DAML to Java code generator (Java codegen) to generate Java types based on a DAML model. You can then use these types in your Java code when reading information from and sending data to the ledger.

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

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).

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

Understand the generated Java model

The Java codegen generates source files in a directory tree under the output directory specified on the command line.

Map DAML primitives to Java types

DAML built-in types are translated to the following equivalent types in Java:

DAML type	Java type	Java Bindings Value Type
`Int`	`java.lang.Long`	Int64
`Numeric`	`java.math.BigDecimal`	Numeric
`Text`	`java.lang.String`	Text
`Bool`	`java.util.Boolean`	Bool
`Party`	`java.lang.String`	Party
`Date`	`java.time.LocalDate`	Date
`Time`	`java.time.Instant`	Timestamp
`List` or `[]`	`java.util.List`	DamlList
`TextMap`	`java.util.Map` Restricted to using `String` keys.	DamlTextMap
`Optional`	`java.util.Optional`	DamlOptional
`()` (Unit)	**None** since the Java language doesn't have a direct equivalent of DAML's Unit type `()`, the generated code uses the Java Bindings value type.	Unit
`ContractId`	Fields of type `ContractId X` refer to the generated `ContractId` class of the respective template `X`.	ContractId

Understand escaping rules

To avoid clashes with Java keywords, the Java codegen applies escaping rules to the following DAML identifiers:

Type names (except the already mapped *built-in types*)
Constructor names
Type parameters
Module names
Field names

If any of these identifiers match one of the Java reserved keywords, the Java codegen appends a dollar sign `$` to the name. For example, a field with the name `import` will be generated as a Java field with the name `import$`.

Understand the generated classes

Every user-defined data type in DAML (template, record, and variant) is represented by one or more Java classes as described in this section.

The Java package for the generated classes is the equivalent of the lowercase DAML module name.

Listing 3: DAML

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

Listing 4: Java

```
package foo.bar.baz;
```

Records (a.k.a product types)

A *DAML record* is represented by a Java class with fields that have the same name as the DAML record fields. A DAML field having the type of another record is represented as a field having the type of the generated class for that record.

Listing 5: Com/Acme/ProductTypes.daml

```
module Com.Acme.ProductTypes where

data Person = Person with name : Name; age : Decimal
data Name = Name with firstName : Text; lastName : Text
```

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

Listing 6: com/acme/producttypes/Person.java

```
package com.acme.producttypes;

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

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

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

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

Listing 7: com/acme/producttypes.Name.java

```
package com.acme.producttypes;

public class Name {
  public final String 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 8: Com/Acme/Templates.daml

```
module Com.Acme.Templates where

data BarKey =
  BarKey
    with
      p : Party
      t : Text

template Bar
  with
    owner: Party
    name: Text
  where
    signatory owner

    key BarKey owner name : BarKey
    maintainer key.p

```

(continues on next page)

(continued from previous page)

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

A file is generated that defines three Java classes:

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

Listing 9: com/acme/templates/Bar.java

```
package com.acme.templates;

public class Bar extends Template {

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

  public final String owner;
  public final String name;

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

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

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

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

  public static class ContractId {
    public final String contractId;

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

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

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

  public static class Contract {
    public final ContractId id;
```

(continues on next page)

(continued from previous page)

```
    public final Bar data;

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

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

Variants (a.k.a sum types)

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

Listing 10: Com/Acme/Variants.daml

```
module Com.Acme.Variants where

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

The Java code generated for this variant is:

Listing 11: com/acme/variants/BookAttribute.java

```
package com.acme.variants;

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

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

Listing 12: com/acme/variants/bookattribute/Pages.java

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

public class Pages extends BookAttribute {
  public final Long longValue;

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

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

Listing 13: com/acme/variants/bookattribute/Authors.java

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

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

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

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

}
```

Listing 14: com/acme/variants/bookattribute/Title.java

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

public class Title extends BookAttribute {
  public final String stringValue;

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

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

Listing 15: com/acme/variants/bookattribute/Published.java

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

public class Published extends BookAttribute {
  public final Long year;
  public final String publisher;

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

  public Published(Long year, String publisher) { /* ... */ }
  public 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 16: Com/Acme/ParametrizedTypes.daml

```
module Com.Acme.ParameterizedTypes where

data Attribute a = Attribute
    with v : a

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

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

Listing 17: com/acme/parametrizedtypes/Attribute.java

```
package com.acme.parametrizedtypes;

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

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

  public 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 18: Com/Acme/Enum.daml

```
module Com.Acme.Enum where

data Color = Red | Blue | Green
```

The Java code generated for this variant is:

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

```
package com.acme.enum;


public enum Color {
```

(continues on next page)

(continued from previous page)

```
  RED,

  GREEN,

  BLUE;

  /* ... */

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

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

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

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

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

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

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

}
```

Convert a value of a generated type to a Java Bindings value

To convert an instance of the generic type `Attribute<a>` to a Java Bindings Value, call the `toValue` method and pass a function as the `toValuea` argument for converting the field of type `a` to the respective Java Bindings Value. The name of the parameter consists of `toValue` and the name of the type parameter, in this case `a`, to form the name `toValuea`.

Below is a Java fragment that converts an attribute with a `java.lang.Long` value to the Java Bindings representation using the *method reference* `Int64::new`.

```
Attribute<Long> pagesAttribute = new Attributes<>(42L);

Value serializedPages = pagesAttribute.toValue(Int64::new);
```

See *DAML To Java Type Mapping* for an overview of the Java Bindings Value types.

Note: If the DAML type is a record or variant with more than one type parameter, you need to pass a conversion function to the `toValue` method for each type parameter.

Create a value of a generated type from a Java Bindings value

Analogous to the `toValue` method, to create a value of a generated type, call the method `fromValue` and pass conversion functions from a Java Bindings Value type to the expected Java type.

```
Attribute<Long> pagesAttribute = Attribute.<Long>fromValue(serializedPages,
    f -> f.asInt64().getOrElseThrow(() -> throw new□
↪IllegalArgumentException("Expected Int field").getValue());
```

See Java Bindings Value class for the methods to transform the Java Bindings types into corresponding Java types.

Non-exposed parameterized types

If the parameterized type is contained in a type where the *actual* type is specified (as in the `BookAttributes` type above), then the conversion methods of the enclosing type provides the required conversion function parameters automatically.

Convert Optional values

The conversion of the Java `Optional` requires two steps. The `Optional` must be mapped in order to convert its contains before to be passed to `DamlOptional::of` function.

```
Attribute<Optional<Long>> idAttribute = new Attribute<List<Long>>(Optional.
↪of(42));

val serializedId = DamlOptional.of(idAttribute.map(Int64::new));
```

To convert back DamlOptional to Java `Optional`, one must use the containers method `toOptional`. This method expects a function to convert back the value possibiy contains in the container.

```
Attribute<Optional<Long>> idAttribute2 =
  serializedId.toOptional(v -> v.asInt64().orElseThrow(() -> new□
↪IllegalArgumentException("Expected Int64 element")));
```

Convert Collection values

DamlCollectors provides collectors to converted Java collection containers such as `List` and `Map` to DamlValues in one pass. The builders for those collectors require functions to convert the element of the container.

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

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

To convert back DAML containers to Java ones, one must use the containers methods `toList` or `toMap`. Those methods expect functions to convert back the container's entries.

```
Attribute<List<String>> authorsAttribute2 =
    Attribute.<List<String>>fromValue(
        serializedAuthors,
        f0 -> f0.asList().orElseThrow(() -> new IllegalArgumentException(
↪"Expected DamlList field"))
```

(continues on next page)

(continued from previous page)

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

3.7.6.2 Example project

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

The former example does not use the Reactive Components, and the latter example does. Both examples implement the PingPong application, which consists of:

a DAML model with two contract templates, Ping and Pong
two parties, Alice and Bob

The logic of the application goes like this:

1. The application injects a contract of type Ping for Alice.
2. Alice sees this contract and exercises the consuming choice RespondPong to create a contract of type Pong for Bob.
3. Bob sees this contract and exercises the consuming choice RespondPing to create a contract of type Ping for Alice.
4. Points 2 and 3 are repeated until the maximum number of contracts defined in the DAML is reached.

Setting up the example projects

To set up the example projects, clone the public GitHub repository at github.com/digital-asset/ex-java-bindings and follow the setup instruction in the README file.

This project contains three examples of the PingPong application, built with gRPC (non-reactive), Reactive and Reactive Component bindings respectively.

Example project

PingPongMain.java

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

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

At high level, the code does the following steps:

creates an instance of DamlLedgerClient connecting to an existing Ledger
connect this instance to the Ledger with DamlLedgerClient.connect()
create two instances of PingPongProcessor, which contain the logic of the automation
(This is where the application reacts to the new Ping or Pong contracts.)
run the PingPongProcessor forever by connecting them to the incoming transactions
inject some contracts for each party of both templates

wait until the application is done

PingPongProcessor.runIndefinitely()

The core of the application is the `PingPongProcessor.runIndefinitely()`.

The `PingPongProcessor` queries the transactions first via the `TransactionsClient` of the `DamlLedgerClient`. Then, for each transaction, it produces `Commands` that will be sent to the Ledger via the `CommandSubmissionClient` of the `DamlLedgerClient`.

Output

The application prints statements similar to these:

```
Bob is exercising RespondPong on #1:0 in workflow Ping-Alice-1 at count 0
Alice is exercising RespondPing on #344:1 in workflow Ping-Alice-7 at count 9
```

The first line shows that:

`Bob` is exercising the `RespondPong` choice on the contract with ID `#1:0` for the workflow `Ping-Alice-1`.
Count `0` means that this is the first choice after the initial `Ping` contract.
The workflow ID `Ping-Alice-1` conveys that this is the workflow triggered by the second initial `Ping` contract that was created by `Alice`.

The second line is analogous to the first one.

3.7.6.3 IOU Quickstart Tutorial

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 *installation*.

On this page:

Download the quickstart application

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

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

Folder structure

The project contains the following files:

```
.
├── daml
│   ├── Iou.daml
│   ├── IouTrade.daml
│   ├── Main.daml
│   ├── Setup.daml
│   └── Tests
│       ├── Iou.daml
│       └── Trade.daml
├── daml.yaml
├── frontend-config.js
├── pom.xml
└── src
    └── main
        ├── java
        │   └── com
        │       └── digitalasset
        │           └── quickstart
        │               └── iou
        │                   └── IouMain.java
        └── resources
            └── logback.xml
```

`daml.yaml` is a DAML project config file used by the SDK to find out how to build the DAML project and how to run it.
`daml` contains the *DAML code* specifying the contract model for the ledger.
`daml/Tests` contains *test scenarios* for the DAML model.
`frontend-config.js` and `ui-backend.conf` are configuration files for the *Navigator* frontend.
`pom.xml` and `src/main/java` constitute a *Java application* that provides REST services to interact with the ledger.

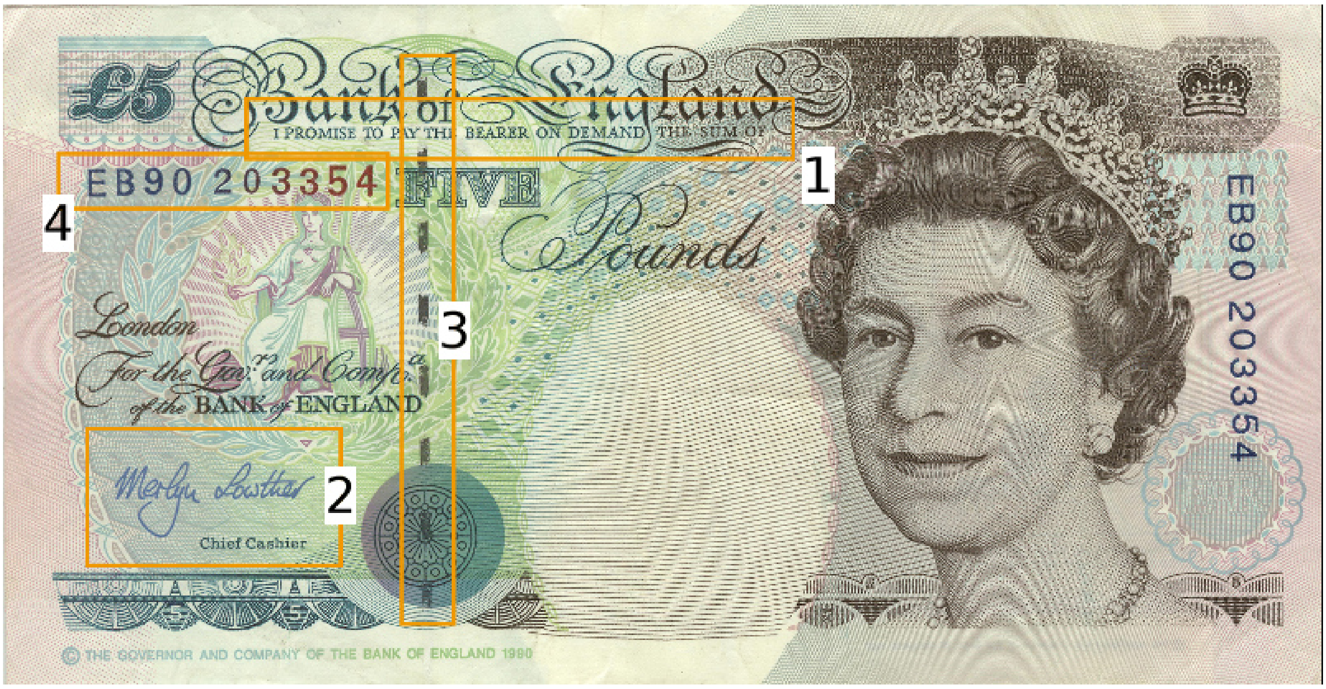
You will explore these in more detail through the rest of this guide.

Overview of what an IOU is

To run through this guide, you will need to understand what an IOU is. This section describes the properties of an IOU like a bank bill that make it useful as a representation and transfer of value.

A bank bill represents a contract between the owner of the bill and its issuer, the central bank. Historically, it is a bearer instrument - it gives anyone who holds it the right to demand a fixed amount of material value, often gold, from the issuer in exchange for the note.

To do this, the note must have certain properties. In particular, the British pound note shown below illustrates the key elements that are needed to describe money in DAML:



1) The Legal Agreement

For a long time, money was backed by physical gold or silver stored in a central bank. The British pound note, for example, represented a promise by the central bank to provide a certain amount of gold or silver in exchange for the note. This historical artifact is still represented by the following statement:

```
I promise to pay the bearer on demand the sum of five pounds.
```

The true value of the note comes from the fact that it physically represents a bearer right that is matched by an obligation on the issuer.

2) The Signature of the Counterparty

The value of a right described in a legal agreement is based on a matching obligation for a counterparty. The British pound note would be worthless if the central bank, as the issuer, did not recognize its obligation to provide a certain amount of gold or silver in exchange for the note. The chief cashier confirms this obligation by signing the note as a delegate for the Bank of England. In general, determining the parties that are involved in a contract is key to understanding its true value.

3) The Security Token

Another feature of the pound note is the security token embedded within the physical paper. It allows the note to be authenticated with limited effort by holding it against a light source. Even a third party can verify the note without requiring explicit confirmation from the issuer that it still acknowledges the associated obligations.

4) The Unique Identifier

Every note has a unique registration number that allows the issuer to track their obligations and detect duplicate bills. Once the issuer has fulfilled the obligations associated with a particular note, duplicates with the same identifier automatically become invalid.

5) The Distribution Mechanism

The note itself is printed on paper, and its legal owner is the person holding it. The physical form of the note allows the rights associated with it to be transferred to other parties that are not explicitly mentioned in the contract.

Run the application using prototyping tools

In this section, you will run the quickstart application and get introduced to the main tools for prototyping DAML:

1. To compile the DAML model, run `daml build`
   This creates a *DAR file* (DAR is just the format that DAML compiles to) called `.daml/dist/quickstart-0.0.1.dar`. The output should look like this:

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

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

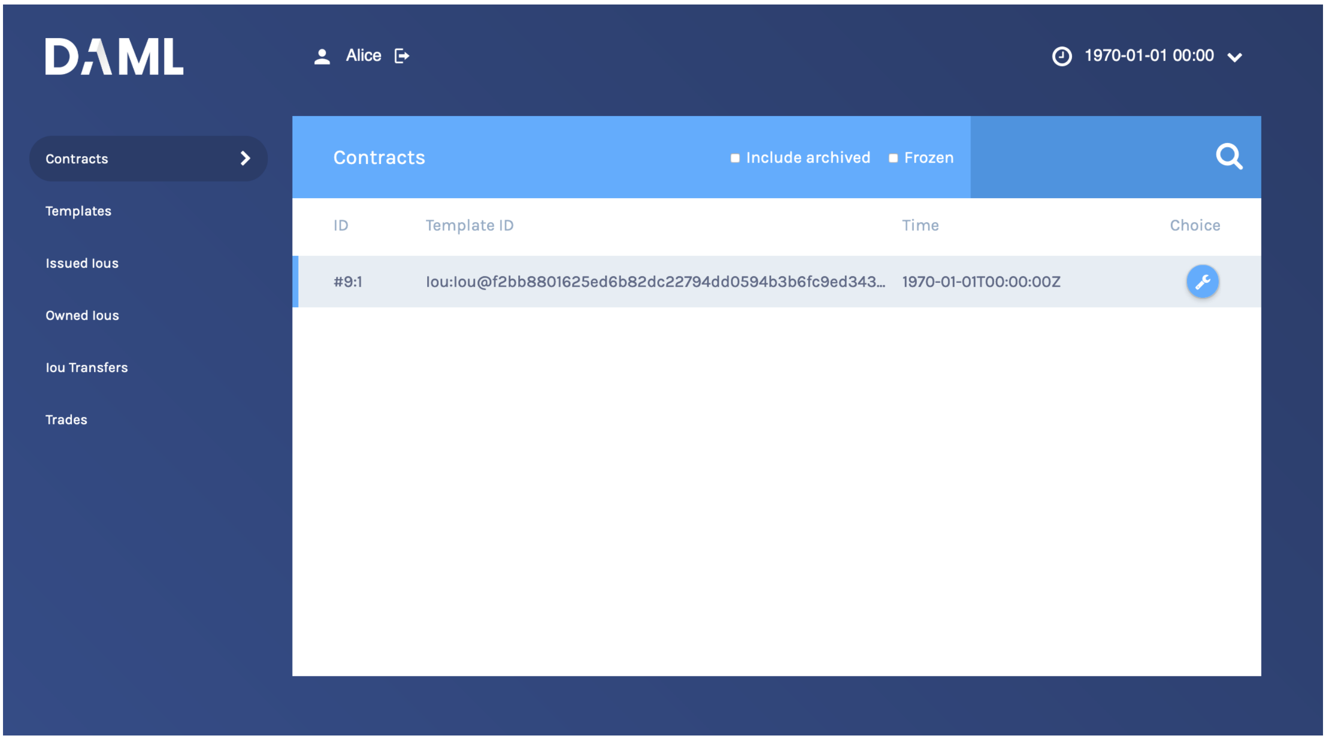
   ```
   DAML LF Engine supports LF versions: 0, 1.0, 1.1, 1.2, 1.3;
    ↪Transaction versions: 1, 2, 3, 4, 5; Value versions: 1, 2, 3, 4
   Starting plainText server
   listening on localhost:6865

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

   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 =
    ↪WallClock, ledger = in-memory, daml-engine = {}
   ```

   The sandbox is now running, and you can access its *ledger API* on port `6865`.
3. Open a new terminal window and navigate to your project directory, `quickstart`.
4. To initialize the ledger with some parties and contracts we use *DAML Script* by running `daml script --dar .daml/dist/quickstart-0.0.1.dar --script-name Setup:initialize --ledger-host localhost --ledger-port 6865 --static-time`
5. 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`.

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:



ID	Template ID	Time	Choice
#9:1	Iou:Iou@f2bb8801625ed6b82dc22794dd0594b3b6fc9ed343...	1970-01-01T00:00:00Z	🔧

   This is showing you what contracts are currently active on the sandbox ledger and visible to *Alice*. You can see that there is a single such contract, in our case with Id `#9:1`, created from a *template* called `Iou:Iou@ffb....`.

   Your contract ID may vary. There's a lot going on in a DAML ledger, so things could have happened in a different order, or other internal ledger events might have occurred. The actual value doesn't matter. We'll refer to this contract as `#9:1` in the rest of this document, and you'll need to substitute your own value mentally.

3. On the left-hand side, you can see what the pages the Navigator contains:

   Contracts
   Templates
   Issued Ious
   Owned Ious
   Iou Transfers
   Trades

   Contracts and **Templates** are standard views, available in any application. The others are created just for this application, specified in the `frontend-config.js` file.

   For information on creating custom Navigator views, see Customizable table views.

4. Click **Templates** to open the Templates page.

   This displays all available *contract templates*. Instances of contracts (or just *contracts*) are created from these templates. The names of the templates are of the format *module.template@hash*. Including the hash disambiguates templates, even when identical module and template names are used between packages.

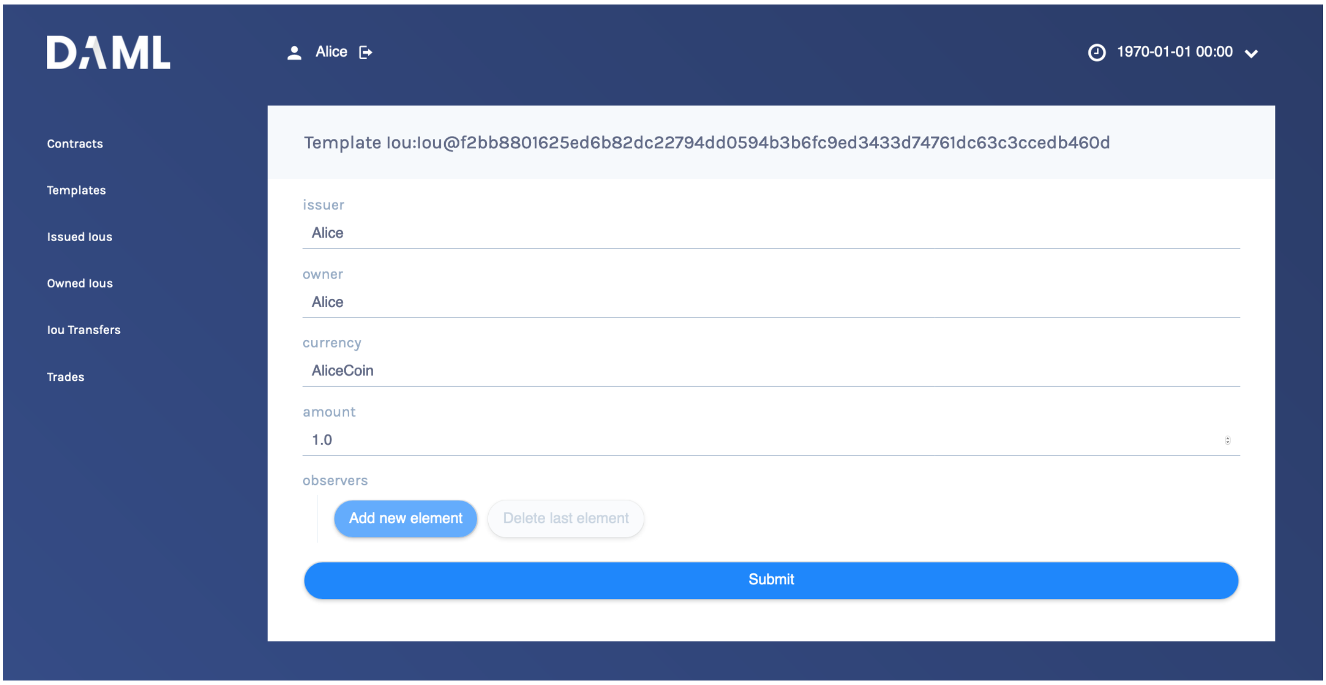
   On the far right, you see the number of *contract instances* that you can see for each template.

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.



Template Iou:Iou@f2bb8801625ed6b82dc22794dd0594b3b6fc9ed3433d74761dc63c3ccedb460d

issuer
Alice

owner
Alice

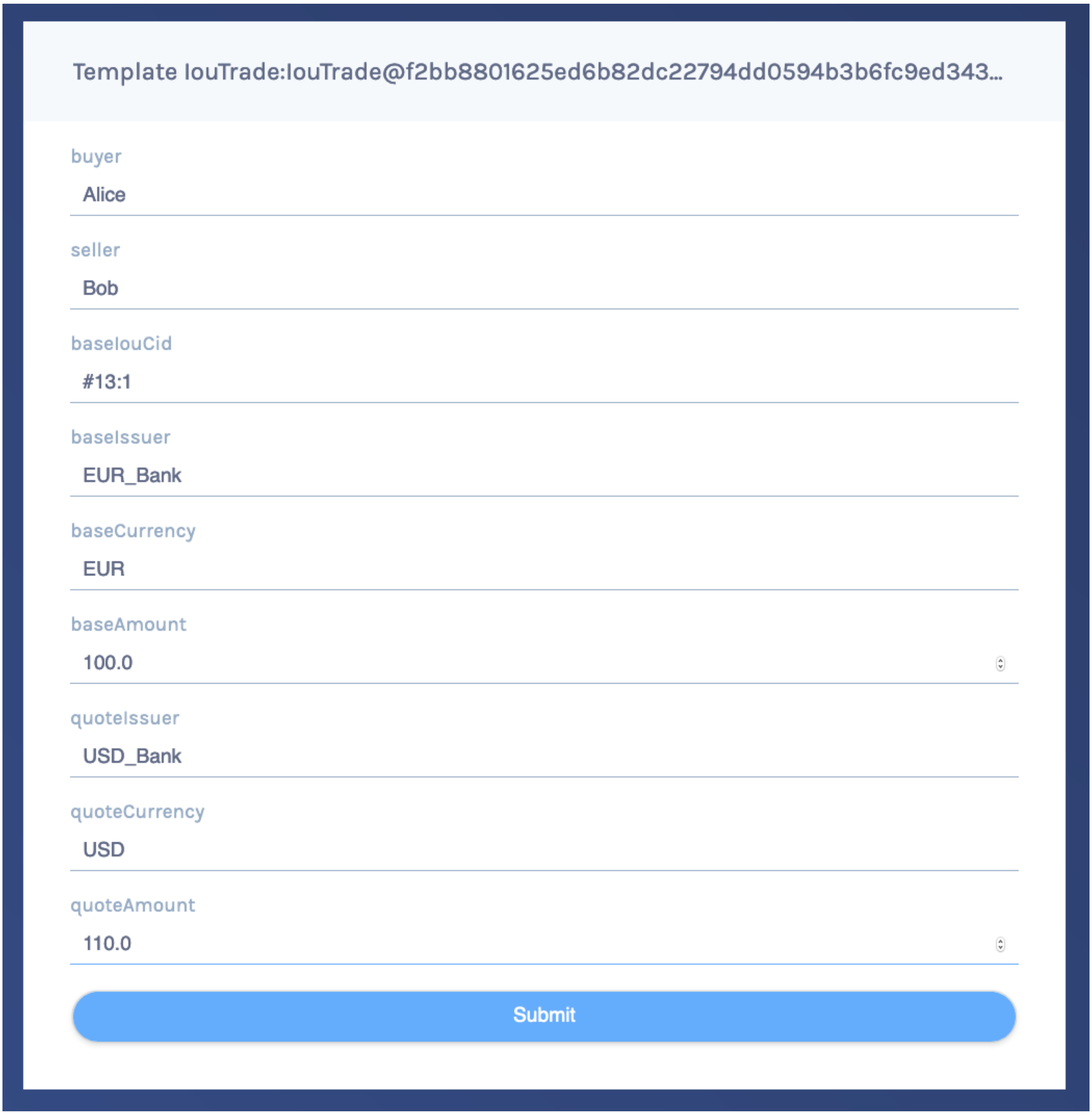
currency
AliceCoin

amount
1.0

observers

The screen shows the same contract `#9:1` that you already saw on the *Contracts* page. It is an Iou for 100, issued by *EUR_Bank*.

9. Go to the **Iou Transfers** page. It shows the transfer of your recently issued Iou to Bob, but Bob has not accepted the transfer, so it is not settled.
   This is an important part of DAML: nobody can be forced into owning an *Iou*, or indeed agreeing to any other contract. They must explicitly consent.
   You could cancel the transfer by using the *IouTransfer_Cancel* choice within it, but for this walk-through, leave it alone for the time being.
10. Try asking *Bob* to exchange your 100 for $110. To do so, you first have to show your Iou to *Bob* so that he can verify the settlement transaction, should he accept the proposal.
    Go back to *Owned Ious*, open the Iou for 100 and click on the button *Iou_AddObserver*. Submit *Bob* as the *newObserver*.
    Contracts in DAML are immutable, meaning they cannot be changed, only created and archived. If you head back to the **Owned Ious** screen, you can see that the Iou now has a new Contract ID. In our case, it's *#13:1*.
11. To propose the trade, go to the **Templates** screen. Click on the *IouTrade:IouTrade* template, fill in the form as shown below and submit the transaction.
12. Go to the **Trades** page. It shows the just-proposed trade.
13. You are now going to switch user to Bob, so you can accept the trades you have just proposed. Start by clicking on the logout button next to the username, at the top of the screen. On the login page, select **Bob** from the dropdown.
14. First, accept the transfer of the *AliceCoin*. Go to the **Iou Transfers** page, click on the row of the transfer, and click **IouTransfer_Accept**, then **Submit**.
15. Go to the **Owned Ious** page. It now shows the *AliceCoin*.
    It also shows an *Iou* for $110 issued by *USD_Bank*. This matches the trade proposal you made earlier as Alice.
    Note its *Contract Id*.
16. Settle the trade. Go to the **Trades** page, and click on the row of the proposal. Accept the trade by clicking **IouTrade_Accept**. In the popup, enter the Contract ID you just noted as the *quoteIouCid*, then click **Submit**.
    The two legs of the transfer are now settled atomically in a single transaction. The trade either



Template IouTrade:IouTrade@f2bb8801625ed6b82dc22794dd0594b3b6fc9ed343...

buyer
Alice

seller
Bob

baseIouCid
#13:1

baseIssuer
EUR_Bank

baseCurrency
EUR

baseAmount
100.0

quoteIssuer
USD_Bank

quoteCurrency
USD

quoteAmount
110.0

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 *DAML Ledger Model*.

    Note: *USD_Bank* does know about an intermediate *IouTransfer* contract that was created and consumed as part of the atomic settlement in the previous step. Since that contract was never active on the ledger, it is not shown in Navigator. You will see how to view a complete transaction graph, including who knows what, in *Test using scenarios* below.

Get started with DAML

The *contract model* specifies the possible contracts, as well as the allowed transactions on the ledger, and is written in DAML.

The core concept in DAML is a *contract template* - you used them earlier to create contracts. Contract templates specify:

- a type of contract that may exist on the ledger, including a corresponding data type
- the *signatories*, who need to agree to the *creation* of a contract 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 DAML Ledgers, consult *DAML Ledger Model* for an in-depth technical description.

Develop with DAML Studio

Take a look at the DAML that specifies the contract model in the quickstart application. The core template is `Iou`.

1. Open *DAML Studio*, a DAML IDE based on VS Code, by running `daml studio` from the root of your project.
2. Using the explorer on the left, open `daml/Iou.daml`.

The first two lines specify language version and module name:

```
module Iou where
```

Next, a template called *Iou* is declared together with its datatype. This template has five fields:

```
template Iou
  with
    issuer : Party
```

(continues on next page)

(continued from previous page)

```
    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
```

(continues on next page)

(continued from previous page)

```
        baseIou <- fetch baseIouCid
        baseIssuer === baseIou.issuer
        baseCurrency === baseIou.currency
        baseAmount === baseIou.amount
        buyer === baseIou.owner
        quoteIou <- fetch quoteIouCid
        quoteIssuer === quoteIou.issuer
        quoteCurrency === quoteIou.currency
        quoteAmount === quoteIou.amount
        seller === quoteIou.owner
        quoteIouTransferCid <- exercise quoteIouCid Iou_Transfer with
          newOwner = buyer
        transferredQuoteIouCid <- exercise quoteIouTransferCid
↪IouTransfer_Accept
        baseIouTransferCid <- exercise baseIouCid Iou_Transfer with
          newOwner = seller
        transferredBaseIouCid <- exercise baseIouTransferCid IouTransfer_
↪Accept
        return (transferredQuoteIouCid, transferredBaseIouCid)
```

This choice uses the === operator from the DAML Standard Library to check pre-conditions. The standard library is imported using `import DA.Assert` at the top of the module.

Then, it *composes* the `Iou_Transfer` and `IouTransfer_Accept` choices to build one big transaction. In this transaction, `buyer` and `seller` exchange their Ious atomically, without disclosing the entire transaction to all parties involved.

The *Issuers* of the two Ious, which are involved in the transaction because they are signatories on the `Iou` and `IouTransfer` contracts, only get to see the sub-transactions that concern them, as we saw earlier.

For a deeper introduction to DAML, consult the *DAML Reference*.

Test using scenarios

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

Scenarios are a linear sequence of transactions that is evaluated using the same consistency, conformance and authorization rules as it would be on the full ledger server or the sandbox ledger. They are integrated into DAML Studio, which can show you the resulting transaction graph, making them a powerful tool to test and troubleshoot the contract model.

To take a look at the scenarios in the quickstart application, open `daml/Tests/Trade.daml` in DAML Studio.

A scenario test is defined with `trade_test = scenario do`. The `submit` function takes a submitting party and a transaction, which is specified the same way as in contract choices.

The following block, for example, issues an `Iou` and transfers it to Alice:

```
-- Banks issue IOU transfers.
iouTransferAliceCid <- submit eurBank do
```

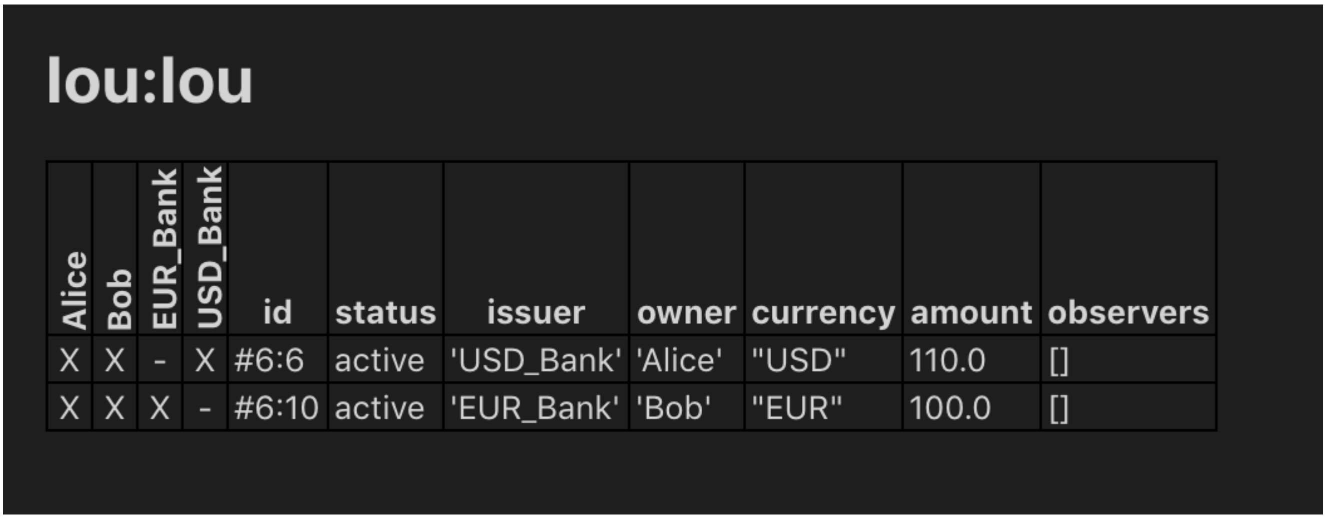
(continues on next page)

(continued from previous page)

```
    createAndExerciseCmd
      Iou with
        issuer = eurBank
        owner = eurBank
        currency = "EUR"
        amount = 100.0
```

Compare the scenario with the `setup` scenario in `daml/Main.daml`. You will see that the scenario you used to initialize the sandbox is an initial segment of the `trade_test` scenario. The latter adds transactions to perform the trade you performed through Navigator, and a couple of transactions in which expectations are verified.

After a short time, the text *Scenario results* should appear above the test. Click on it to open the visualization of the resulting ledger state.



Iou:Iou

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

Each row shows a contract on the ledger. The first four columns show which parties know of which contracts. The remaining columns show the data on the contracts. You can see past contracts by checking the **Show archived** box at the top. Click the adjacent **Show transaction view** button to switch to a view of the entire transaction tree.

In the transaction view, transaction `#6` is of particular interest, as it shows how the Ious are exchanged atomically in one transaction. The lines starting `known to (since)` show that the Banks do indeed not know anything they should not:

```
TX #6 1970-01-01T00:00:00Z (Tests.Trade:61:14)
#6:0
│   known to (since): 'Alice' (#6), 'Bob' (#6)
└─> 'Bob' exercises IouTrade_Accept on #5:0 (IouTrade:IouTrade)
          with
            quoteIouCid = #3:1
    children:
    #6:1
    │   known to (since): 'Alice' (#6), 'Bob' (#6)
    └─> fetch #4:1 (Iou:Iou)

    #6:2
```

(continues on next page)

(continued from previous page)

```
    │   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
                   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
```

(continues on next page)

(continued from previous page)

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

The `submit` function used in this scenario tries to perform a transaction and fails if any of the ledger integrity rules are violated. There is also a `submitMustFail` function, which checks that certain transactions are not possible. This is used in `daml/Tests/Iou.daml`, for example, to confirm that the ledger model prevents double spends.

Integrate with the ledger

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

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

To compile the Java integration for the quickstart application, we first need to run the Java codegen on the DAR we built before:

```
daml codegen java
```

Once the code has been generated, we can now compile it using `mvn compile`.

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

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

Note: To start the same application on another port, use the command-line parameter `-Drestport=PORT`. To start it for another party, use `-Dparty=PARTY`.

For example, to start the application for Bob on 8081, run `mvn exec:java@run-quickstart -Drestport=8081 -Dparty=Bob`

The following REST services are included:

`GET` on `http://localhost:8080/iou` lists all active Ious, and their Ids.
Note that the Ids exposed by the REST API are not the ledger contract Ids, but integers. You can open the address in your browser or run `curl -X GET http://localhost:8080/iou`.

`GET` on `http://localhost:8080/iou/ID` returns the Iou with Id `ID`.
For example, to get the content of the Iou with Id 0, run:
`curl -X GET http://localhost:8080/iou/0`

`PUT` on `http://localhost:8080/iou` creates a new Iou on the ledger.
To create another *AliceCoin*, run:
`curl -X PUT -d '{"issuer":"Alice","owner":"Alice", "currency":"AliceCoin","amount":1.0,"observers":[]}' http://localhost:8080/iou`

`POST` on `http://localhost:8080/iou/ID/transfer` transfers the Iou with Id `ID`.
Check the Id of your new *AliceCoin* by listing all active Ious. If you have followed this guide, it will be `0` so you can run:
`curl -X POST -d '{ "newOwner":"Bob" }' http://localhost:8080/iou/0/transfer`
to transfer it to Bob. If it's not `0`, just replace the `0` in `iou/0` in the above command.

The automation is based on the *Java bindings* and the output of the *Java code generator*, which are included as a Maven dependency and Maven plugin respectively:

```xml
        <dependency>
            <groupId>com.daml</groupId>
            <artifactId>bindings-rxjava</artifactId>
            <version>__VERSION__</version>
            <exclusions>
                <exclusion>
                    <groupId>com.google.protobuf</groupId>
                    <artifactId>protobuf-lite</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
```

It consists of the application in file `IouMain.java`. It uses the class `Iou` from `Iou.java`, which is generated from the DAML model with the Java code generator. The `Iou` class provides better serialization and de-serialization to JSON via gson.

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

```java
// 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,
            Optional.empty(),
            Optional.empty(),
            Optional.empty(),
            Collections.singletonList(c))
            .blockingGet();
}
```

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

Listing 21: ExerciseCommand

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

Listing 22: 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*.

Next steps

Great - you've completed the quickstart guide!

Some steps you could take next include:

- Explore *examples* for guidance and inspiration.
- *Learn DAML*.
- *Language reference*.
- Learn more about *application development*.
- Learn about the *conceptual models* behind DAML.

The Java bindings is a client implementation of the *Ledger API* based on RxJava, a library for composing asynchronous and event-based programs using observable sequences for the Java VM. It provides an idiomatic way to write DAML Ledger applications.

See also:

This documentation for the Java bindings API includes the JavaDoc reference documentation.

3.7.6.4 Overview

The Java bindings library is composed of:

The Data Layer A Java-idiomatic layer based on the Ledger API generated classes. This layer simplifies the code required to work with the Ledger API.
Can be found in the java package `com.daml.ledger.javaapi.data`.

The Reactive Layer A thin layer built on top of the Ledger API services generated classes.
For each Ledger API service, there is a reactive counterpart with a matching name. For instance, the reactive counterpart of `ActiveContractsServiceGrpc` is `ActiveContractsClient`.
The Reactive Layer also exposes the main interface representing a client connecting via the Ledger API. This interface is called `LedgerClient` and the main implementation working against a DAML Ledger is the `DamlLedgerClient`.
Can be found in the java package `com.daml.ledger.rxjava`.

The Reactive Components A set of optional components you can use to assemble DAML Ledger applications.
The most important components are:

- the `LedgerView`, which provides a local view of the Ledger
- the `Bot`, which provides utility methods to assemble automation logic for the Ledger

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

Code generation

When writing applications for the ledger in Java, you want to work with a representation of DAML templates and data types in Java that closely resemble the original DAML code while still being as true to the native types in Java as possible.

To achieve this, you can use DAML to Java code generator (Java codegen) to generate Java types based on a DAML model. You can then use these types in your Java code when reading information from and sending data to the ledger.

For more information on Java code generation, see *Generate Java code from DAML*.

Connecting to the ledger: LedgerClient

Connections to the ledger are made by creating instance of classes that implement the interface `LedgerClient`. The class `DamlLedgerClient` implements this interface, and is used to connect to a DAML ledger.

This class provides access to the ledgerId, and all clients that give access to the various ledger services, such as the active contract set, the transaction service, the time service, etc. This is described *below*. Consult the JavaDoc for DamlLedgerClient for full details.

Accessing data on the ledger: LedgerView

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

Note:

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

The `LedgerView` is updated every time:

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

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

Writing automations: Bot

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

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

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

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

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

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

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

In the above:

`applicationId` The id used by the Ledger to identify all the queries from the same application.

`ledgerClient` The connection to the Ledger.

`transactionFilter` The server-side filter to the incoming transactions. Used to reduce the traffic between Ledger and application and make an application more efficient.

`bot` The logic of the application,

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

3.7.6.5 Reference documentation

Click here for the JavaDoc reference documentation.

3.7.6.6 Getting started

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

Set up a Maven project

To use the Java bindings library, add the following dependencies to your project's pom.xml:

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

Replace x.y.z for both dependencies with the version that you want to use. You can find the available versions by checking the Maven Central Repository.

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*.

Connecting to the ledger

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

Authorizing

Some ledgers will require you to send an access token along with each request.

To learn more about authorization, read the Authorization overview.

To use the same token for all Ledger API requests, the DamlLedgerClient builders expose a withAccessToken method. This will allow you to not pass a token explicitly for every call.

If your application is long-lived and your tokens are bound to expire, you can reload the necessary token when needed and pass it explicitly for every call. Every client method has an overload that allows a token to be passed, as in the following example:

```
transactionClient.getLedgerEnd(); // Uses the token specified when
↪constructing the client
transactionClient.getLedgerEnd(accessToken); // Override the token for
↪this call exclusively
```

If you're communicating with a ledger that verifies authorization it's very important to secure the communication channel to prevent your tokens to be exposed to man-in-the-middle attacks. The next chapter describes how to enable TLS.

Connecting securely

The Java bindings library lets you connect to a DAML Ledger via a secure connection. The builders created by `DamlLedgerClient.newBuilder` default to a plaintext connection, but you can invoke `withSslContext` to pass an ``SslContext`. Using the default plaintext connection is useful only when connecting to a locally running Sandbox for development purposes.

Secure connections to a DAML Ledger must be configured to use client authentication certificates, which can be provided by a Ledger Operator.

For information on how to set up an `SslContext` with the provided certificates for client authentication, please consult the gRPC documentation on TLS with OpenSSL as well as the HelloWorldClientTls example of the `grpc-java` project.

Advanced connection settings

Sometimes the default settings for gRPC connections/channels are not suitable for a given situation. These use cases are supported by creating a a custom NettyChannelBuilder object and passing the it to the `newBuilder` static method defined over DamlLedgerClient.

3.7.6.7 Example project

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

3.7.7 Scala bindings

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

3.7.7.1 Introduction

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

There are two main components:

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

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

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

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

3.7.7.2 Getting started

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

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

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

lazy val applicationDependencies = Seq(
  "com.daml" %% "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`)
```

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

4) Run Scala codegen:

```
$ daml codegen scala
```

If the command is successful, it should print:

```
Scala codegen
Reading configuration from project configuration file
```

(continues on next page)

(continued from previous page)

```
[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).

3.7.7.4 Example code

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

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

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

Create a contract and send a CreateCommand

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

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

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

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

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

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

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

```
    createCmd = iou.create
    _ <- clientUtil.submitCommand(issuer, issuerWorkflowId, createCmd)
```

(continues on next page)

(continued from previous page)

```
 _ = logger.info(s"$issuer created IOU: $iou")
 _ = logger.info(s"$issuer sent create command: $createCmd")
```

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

Receive a transaction, exercise a choice and send an ExerciseCommand

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

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

and the following code that handles subscription and decoding:

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

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

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

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

3.7.7.5 Authorization

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

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

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

(continues on next page)

(continued from previous page)

```
    sslContext = None,
    token = None
  )
```

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

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

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

3.7.8 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.

3.7.9 Creating your own bindings

This page gets you started with creating custom bindings for a DAML Ledger.

Bindings for a language consist of two main components:

Ledger API Client stubs for the programming language, – the remote API that allows sending ledger commands and receiving ledger transactions. You have to generate **Ledger API** from the gRPC protobuf definitions in the daml repository on GitHub. **Ledger API** is documented on this page: *gRPC*. The gRPC tutorial explains how to generate client stubs .

Codegen A code generator is a program that generates classes representing DAML contract templates in the language. These classes incorporate all boilerplate code for constructing: *CreateCommand* and *ExerciseCommand* corresponding for each DAML contract template.

Technically codegen is optional. You can construct the commands manually from the auto-generated **Ledger API** classes. However, it is very tedious and error-prone. If you are creating *ad hoc* bindings for a project with a few contract templates, writing a proper codegen may be overkill. On the other hand, if you have hundreds of contract templates in your project or are planning to build language bindings that you will share across multiple projects, we recommend including a codegen in your bindings. It will save you and your users time in the long run.

Note that for different reasons we chose codegen, but that is not the only option. There is really a broad category of metaprogramming features that can solve this problem just as well or even better than codegen; they are language-specific, but often much easier to maintain (i.e. no need to add a build step). Some examples are:

F# Type Providers
Template Haskell
Scala macro annotations (not future-proof enough to use when implementing the last Scala codegen)

3.7.9.1 Building Ledger Commands

No matter what approach you take, either manually building commands or writing a codegen to do this, you need to understand how ledger commands are structured. This section demonstrates how to build create and exercise commands manually and how it can be done using contract classes generated by Scala codegen.

Create Command

Let's recall an **IOU** example from the *Quickstart guide*, where *Iou* template is defined like this:

```
template Iou
  with
    issuer : Party
    owner : Party
    currency : Text
    amount : Decimal
    observers : [Party]
```

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

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

If you do not specify any of the above fields or type their names or values incorrectly, or do not order them exactly as they are in the DAML template, the above code will compile but fail at run-time because you did not structure your create command correctly.

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

```
  val iou = M.Iou(
    issuer = issuer,
    owner = issuer,
```

(continues on next page)

(continued from previous page)

```
    currency = "USD",
    amount = BigDecimal("1000.00"),
    observers = List())
```

Exercise Command

To build *ExerciseCommand* for *Iou_Transfer*:

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

manually in Scala:

```
  def iouTransferExerciseCommand(
      templateId: Identifier,
      contractId: String,
      newOwner: String): Command.Exercise = {
    val transferTemplateId = Identifier(
      packageId = templateId.packageId,
      moduleName = templateId.moduleName,
      entityName = "Iou_Transfer")
    val fields = Seq(RecordField("newOwner", Some(Value(Value.Sum.
↪Party(newOwner)))))
    Command.Exercise(
      ExerciseCommand(
        templateId = Some(templateId),
        contractId = contractId,
        choice = "Iou_Transfer",
        choiceArgument = Some(Value(Value.Sum.
↪Record(Record(Some(transferTemplateId), fields))))
      ))
  }
```

versus creating the same command using a value class generated by *Scala codegen*:

```
    exerciseCmd = iouContract.contractId.exerciseIou_Transfer(actor =□
↪issuer, newOwner = newOwner)
```

3.7.9.2 Summary

When creating custom bindings for DAML Ledgers, you will need to:

generate **Ledger API** from the gRPC definitions

decide whether to write a codegen to generate ledger commands or manually build them for all contracts defined in your DAML model.

The above examples should help you get started. If you are creating custom binding or have any questions, see the *Support* page for how to get in touch with us.

3.7.9.3 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/

Documentation for Protobuf well known types : https://developers.google.com/protocol-buffers/docs/reference/google.protobuf

DAML Ledger API gRPC Protobuf definitions

- current master: https://github.com/digital-asset/daml/tree/master/ledger-api/grpc-definitions
- for specific versions: https://github.com/digital-asset/daml/releases

Required gRPC Protobuf definitions:

- https://raw.githubusercontent.com/grpc/grpc/v1.18.0/src/proto/grpc/status/status.proto
- https://raw.githubusercontent.com/grpc/grpc/v1.18.0/src/proto/grpc/health/v1/health.proto

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

3.7.10 What's in the Ledger API

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

Submitting commands to the ledger

- Use the *command submission service* to submit commands (create a contract or exercise a choice) to the ledger.
- Use the *command completion service* to track the status of submitted commands.
- Use the *command service* for a convenient service that combines the command submission and completion services.

Reading from the ledger

- Use the *transaction service* to stream committed transactions and the resulting events (choices exercised, and contracts created or archived), and to look up transactions.
- Use the *active contracts service* to quickly bootstrap an application with the currently active contracts. It saves you the work to process the ledger from the beginning to obtain its current state.

Utility services

- Use the *package service* to query the DAML packages deployed to the ledger.
- Use the *ledger identity service* to retrieve the Ledger ID of the ledger the application is connected to.
- Use the *ledger configuration service* to retrieve some dynamic properties of the ledger, like maximum deduplication time for commands.

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.

3.7.11 DAML-LF

When you *compile DAML source into a .dar file*, the underlying format is DAML-LF. DAML-LF is similar to DAML, but is stripped down to a core set of features. The relationship between the surface DAML syntax and DAML-LF is loosely similar to that between Java and JVM bytecode.

As a user, you don't need to interact with DAML-LF directly. But 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)

3.7.11.1 When you need to know about DAML-LF

DAML-LF is only really relevant when you're dealing with the objects you send to or receive from the ledger. If you use any of the provided language bindings for the Ledger API, you don't need to know about DAML-LF at all, because this generates idiomatic representations of DAML for you.

Otherwise, it can be helpful to know what the types in your DAML code look like at the DAML-LF level, so you know what to expect from the Ledger API.

For example, if you are writing an application that creates some DAML contracts, you need to construct values to pass as parameters to the contract. These values are determined by the DAML-LF types in that contract template. This means you need an idea of how the DAML-LF types correspond to the types in the original DAML model.

For the most part the translation of types from DAML to DAML-LF should not be surprising. *This page goes through all the cases in detail*.

For the bindings to your specific programming language, you should refer to the language-specific documentation.

Chapter 4

Deploying to DAML ledgers

4.1 Overview of DAML ledgers

This is an overview of DAML deployment options. Instructions on how to deploy to a specific ledger are available in the following section.

4.1.1 Commercial Integrations

The following table lists commercially supported DAML ledgers and environments that are available for production use today.

Product	Ledger	Vendor
DAML on Corda	Corda	Multiple. Contact Digital Asset
Sextant for DAML	Amazon Aurora	Blockchain Technology Partners
Sextant for DAML	Hyperledger Sawtooth	Blockchain Technology Partners
Sextant for DAML	Amazon QLDB	Blockchain Technology Partners
project : DABL	Managed cloud enviroment	Digital Asset

4.1.2 Open Source Integrations

The following table lists open source DAML integrations.

Ledger	Developer	More Information
Hyperledger Sawtooth	Blockchain Technology Partners	Github Repo
Hyperledger Fabric	Digital Asset	Github Repo
PostgreSQL	Digital Asset	DAML Sandbox Docs

4.1.3 DAML Ledgers in Development

The following table lists the ledgers that are implementing support for running DAML.

Ledger	Developer	More Information
VMware Blockchain	VMware	Press release, April 2019
Hyperledger Besu	Blockchain Technology Partners	Press release, March 2020
FISCO BCOS	WeBank	Press release, April 2020
Canton	Digital Asset reference implementation	canton.io

4.2 Deploying to a generic DAML ledger

DAML ledgers expose a unified administration API. This means that deploying to a DAML ledger is no different from deploying to your local sandbox.

To deploy to a DAML ledger, run the following command from within your DAML project:

```
$ daml deploy --host=<HOST> --port=<PORT> --access-token-file=<TOKEN-FILE>
```

where `<HOST>` and `<PORT>` is the hostname and port your ledger is listening on, which defaults to port `6564`. The `<TOKEN-FILE>` is needed if your sandbox runs with authorization and needs to contain a JWT token with an `admin` claim. If your sandbox is not setup to use any authentication it can be omitted.

Instead of passing `--host` and `--port` flags to the command above, you can add the following section to the project's `daml.yaml` file:

```
ledger:
    host: <HOSTNAME>
    port: <PORT>
```

The `daml deploy` command will

1. upload the project's compiled DAR file to the ledger. This will make the DAML templates defined in the current project available to the API users of the sandbox.
2. allocate the parties specified in the project's `daml.yaml` on the ledger if they are missing.

For more further interactions with the ledger, use the `daml ledger` command. Try running `daml ledger --help` to get a list of available ledger commands:

```
$ daml ledger --help
Usage: daml ledger COMMAND
  Interact with a remote DAML ledger. You can specify the ledger in daml.
↪yaml
  with the ledger.host and ledger.port options, or you can pass the --host␣
↪and
  --port flags to each command below. If the ledger is authenticated, you␣
↪should
  pass the name of the file containing the token using the --access-token-
↪file
  flag.

Available options:
  -h,--help                Show this help text

Available commands:
  list-parties             List parties known to ledger
  allocate-parties         Allocate parties on ledger
  upload-dar               Upload DAR file to ledger
  navigator                Launch Navigator on ledger
```

4.2.1 Connecting via TLS

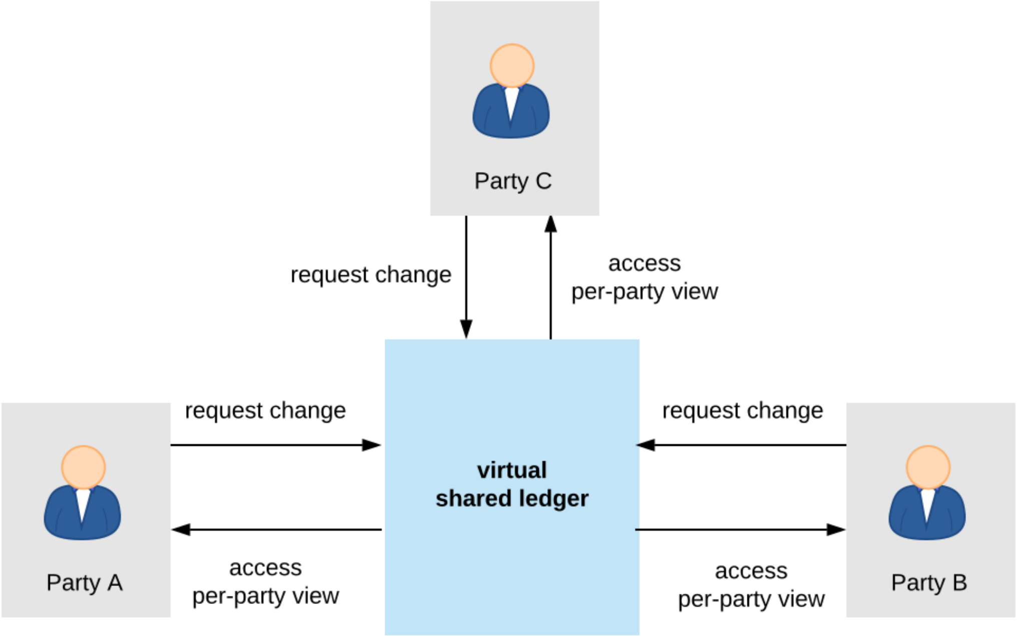
To connect to the ledger via TLS, you can pass `--tls` to the various commands. If your ledger supports or requires mutual authentication you can pass your client key and certificate chain files via `--pem client_key.pem --crt client.crt`. Finally, you can use a custom certificate authority for validating the server certificate by passing `--cacrt server.crt`. If `--pem`, `--crt` or `--cacrt` are specified TLS is enabled automatically so `--tls` is redundant.

4.2.2 Configuring Request Timeouts

You can configure the timeout used on API requests by passing `--timeout=N` to the various `daml ledger` commands and `daml deploy` which will set the timeout to N seconds. Note that this is a per-request timeout not a timeout for the whole command. That matters for commands like `daml deploy` that consist of multiple requests.

4.3 DAML Ledger Topologies

The Ledger API provides parties with an abstraction of a virtual shared ledger, visualized as follows.



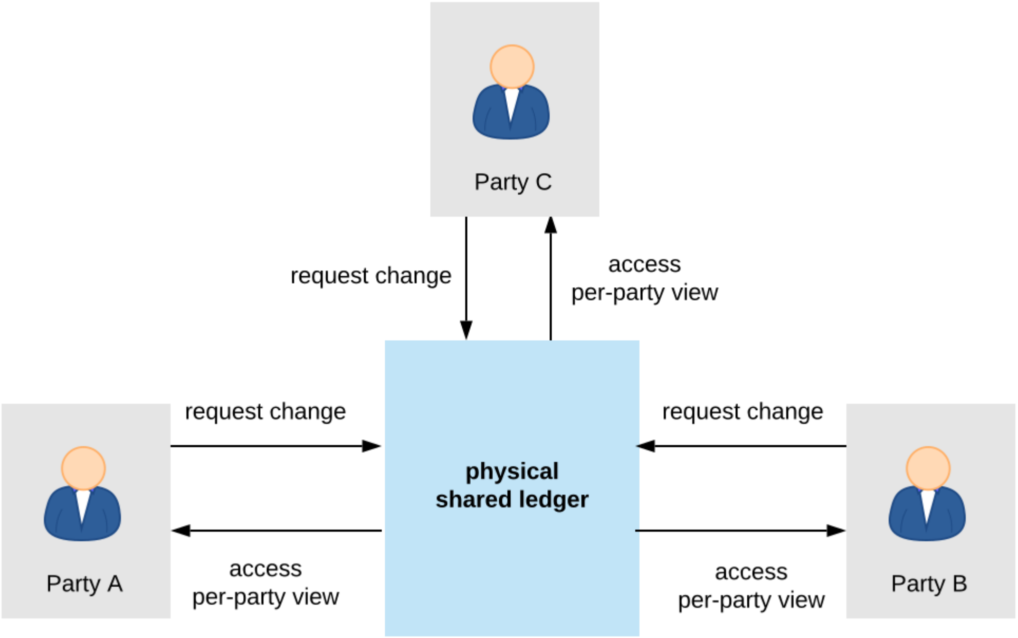
The real-world topologies of actual ledger implementations differ significantly, however. The topologies can impact both the functional and non-functional properties of the resulting ledger. This document provides one useful categorization of the existing implementations' topologies: the split into global and partial state topologies, depending on whether single *trust domains* can see the entire ledger, or just parts of it. The implementations with topologies from the same category share many non-functional properties and trust assumptions. Additionally, their identity and package management functions also behave similarly.

4.3.1 Global State Topologies

In global state topologies, there exists at least one *trust domain* whose systems contain a physical copy of the entire virtual shared ledger that is accessible through the API.

4.3.1.1 The Fully Centralized Ledger

The simplest global state topology is the one where the virtual shared ledger is implemented through a single machine containing a physical copy of the shared ledger, whose real-world owner is called the **operator**.



The *DAML Sandbox* uses this topology. While simple to deploy and operate, the single-machine setup also has downsides:

1. it provides no scaling
2. it is not highly available
3. the operator is fully trusted with preserving the ledger's integrity
4. the operator has full insight into the entire ledger, and is thus fully trusted with privacy
5. it provides no built-in way to interoperate (transactionally share data) across several deployed ledgers; each deployment defines its own segregated virtual shared ledger.

The first four problems can be solved or mitigated as follows:

1. scaling by splitting the system up into separate functional components and parallelizing execution
2. availability by replication
3. trust for integrity by introducing multiple trust domains and distributing trust using Byzantine fault tolerant replication, or by maintaining one trust domain but using hardware-based Trusted Execution Environments (TEEs) or other cryptographic means to enforce or audit ledger integrity without having to trust the operator.
4. trust for privacy through TEEs that restrict data access by hardware means.

The remainder of the section discusses these solutions and their implementations in the different DAML ledgers. The last problem, interoperability, is inherent when the two deployments are operated by different trust domains: by definition, a topology in which no single trust domain would hold the entire ledger is not a global state topology.

4.3.1.2 Scaling

The main functionalities of a system providing the Ledger API are:

1. serving the API itself (handling the gRPC connections, authenticating users, etc),
2. allowing the API users to access their *ledger projection* (reading the ledger), and
3. allowing the API users to issue commands and thus attempt to append commits to the shared ledger (writing to the ledger).

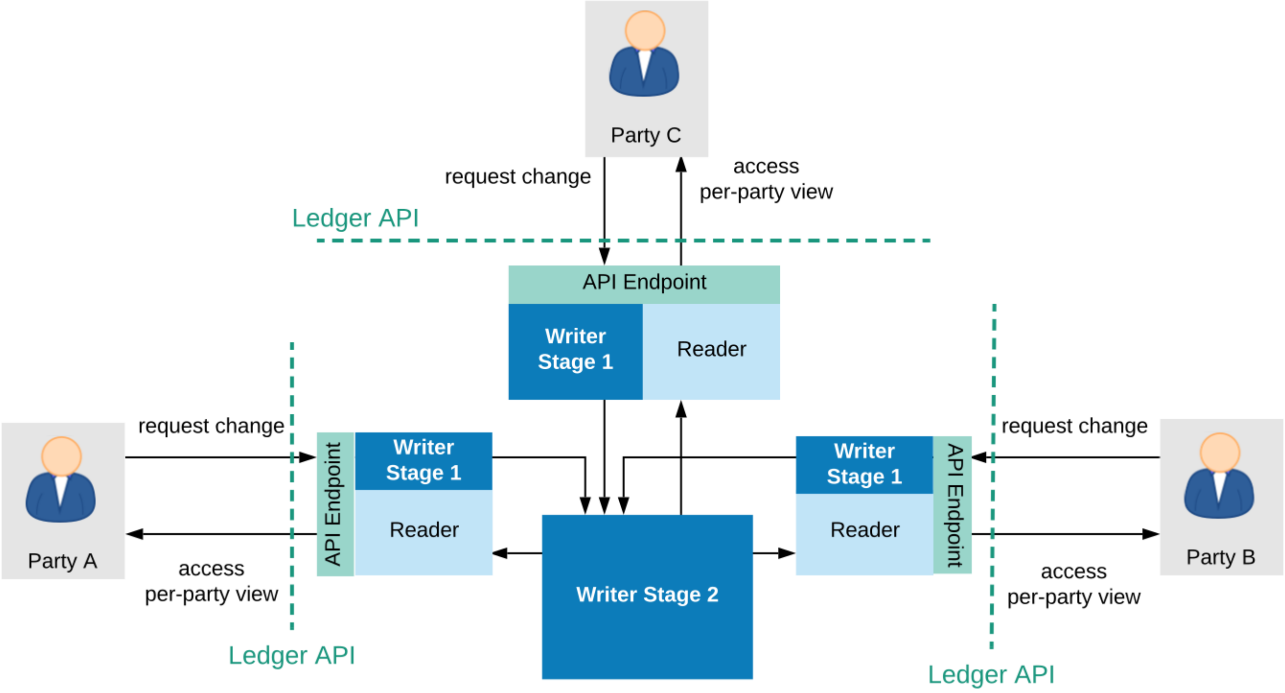
The implementation thus naturally splits up into components for serving the API, reading from the ledger, and writing to the ledger. Serving the API and reading can be scaled out horizontally. Reading can be scaled out by building caches of the ledger contents; as the projections are streams, no synchronization between the different caches is necessary.

To ensure ledger integrity, the writing component must preserve the ledger's *validity conditions*. Writing can thus be further split up into three sub-components, one for each of the three validity conditions:

1. *model conformance* checks (i.e., DAML intepretation),
2. *authorization* checks, and
3. *consistency* checks.

Of these three, conformance and authorization checks can be checked in isolation for each commit. Thus, such checks can be parallelized and scaled out. The consistency check cannot be done in isolation and requires synchronization. However, to improve scaling, it can internally still use some form of sharding, together with a commit protocol.

For example, the next versions of DAML on Amazon Aurora and on Hyperledger Fabric will use such partitioned topologies. The next image shows an extreme version of this partitioning, where each party is served by a separate system node running all the parallelizable functions. The writing subsystem is split into two stages. The first stage checks conformance and authorization, and can be arbitrarily replicated, while the second stage is centralized and checks consistency.



4.3.1.3 Replication: Availability and Distributing Trust

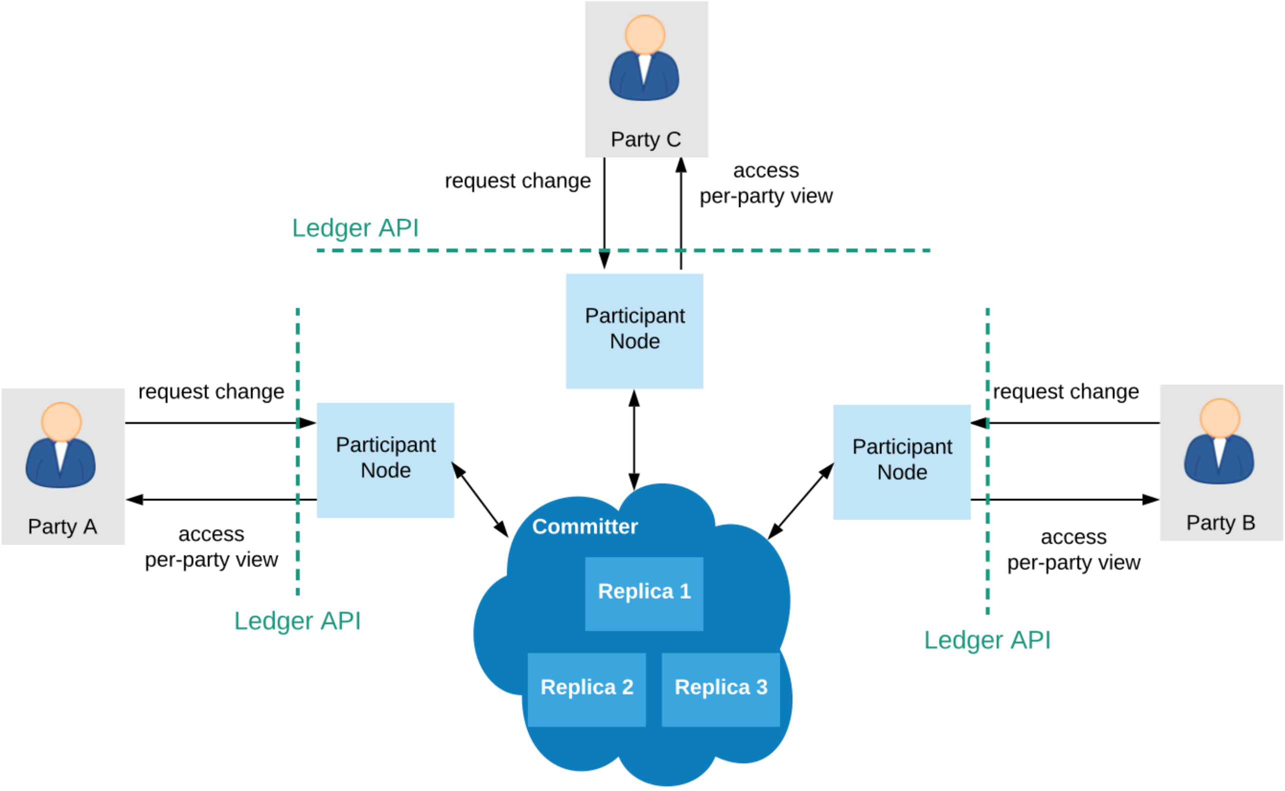
Availability is improved by replication. The scaling methodology described in the previous section already improves the ledger's availability properties, as it introduces replication for most functions.

For example, if a node serving a client with the API fails, clients can fail over to other such nodes. Replicating the writer's consistency-checking subsystem must use a consensus algorithm to ensure consistency of the replicated system (in particular, the linearizability of the virtual shared ledger).

Replication can also help to lower, or more precisely distribute the trust required to ensure the system's integrity. Trust can be distributed by introducing multiple organizations, i.e., multiple trust domains into the system. In these situations, the system typically consists of two types of nodes:

1. **Writer nodes**, which replicate the physical shared ledger and can extend it with new commits. Writer nodes are thus also referred to as **committer nodes**.
2. **Participant nodes**, (also called Client nodes in some platforms) which serve the Ledger API to a subset of the system parties, which we say are hosted by this participant. A participant node proposes new commits on behalf of the parties it hosts, and holds a portion of the ledger that is relevant for those parties (i.e., the parties' *ledger projection*). The term participant node is sometimes also used more generally, for any physical node serving the Ledger API to a party.

The participant nodes need not be trusted by the other nodes, or by the committer(s); the participants can be operated by mutually distrusting entities, i.e., belong to different trust domains. In general, the participant nodes do not necessarily even need to know each other. However, they have to be known to and accepted by the committer nodes. The committer nodes are jointly trusted with ensuring the ledger's integrity. To distribute the trust, the committer nodes must implement a Byzantine fault tolerant replication mechanism. For example, the mechanism can ensure that the system preserves integrity even if up to a third of the committer nodes (e.g., 2 out of 7) misbehave in arbitrary ways. The resulting topology is visualized below.



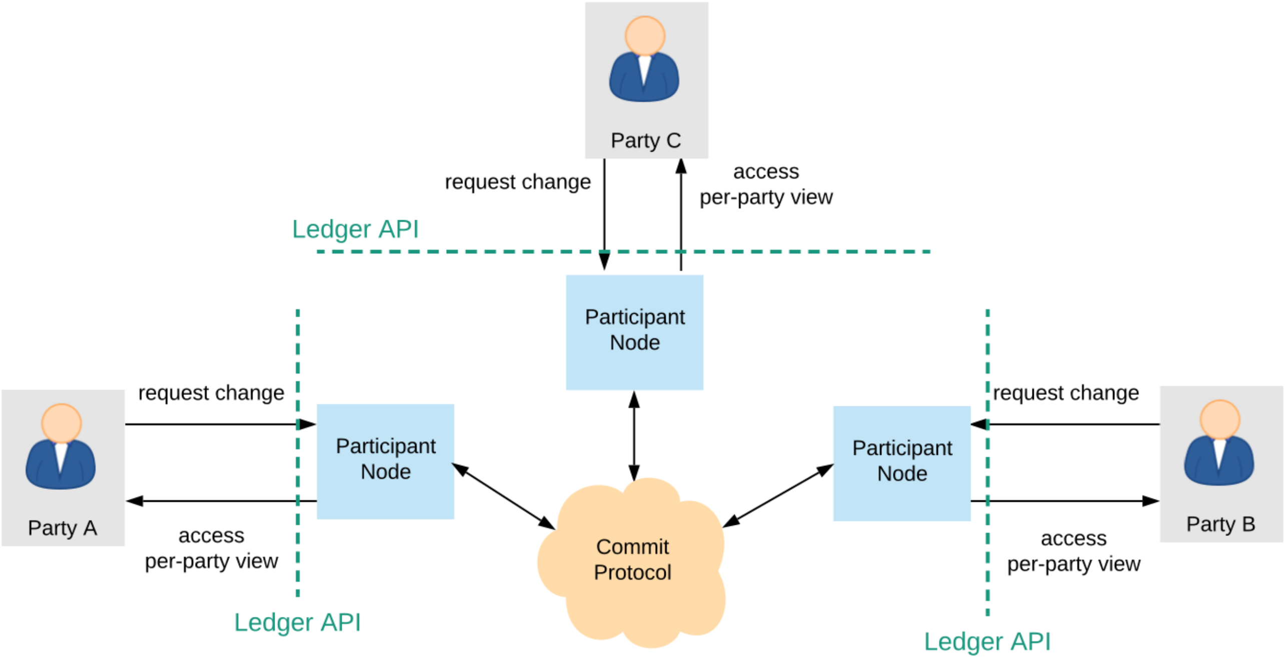
DAML on VMware Concord and DAML on Hyperledger Sawtooth are examples of such a replicated setup.

4.3.1.4 Trusted Execution Environments

Integrity and privacy can also be protected using hardware Trusted Execution Environments (TEEs), such as Intel SGX. The software implementing the ledger can then be deployed inside of TEE **enclaves**, which are code blocks that the processor isolates and protects from the rest of the software stack (even the operating system). The hardware ensures that the enclave data never leaves the processor unencrypted, offering privacy. Furthermore, hardware-based attestation can guarantee that the operating entities process data using the prescribed code only, guaranteeing integrity. The hardware is designed in such a way as to make any potential physical attacks by the operator extremely expensive. This moves the trust necessary to achieve these properties from the operators of the trust domains that maintain the global state to the hardware manufacturer, who is anyway trusted with correctly producing the hardware. Recent security research has, however, found scenarios where the TEE protection mechanisms can be compromised.

4.3.2 Partitioned Ledger Topologies

In these topologies, the ledger is implemented as a distributed system. Unlike the global state topologies, no single trust domain holds a physical copy of the entire shared ledger. Instead, the participant nodes hold just the part of the ledger (i.e., the *ledger projection*) that is relevant to the parties to whom they serve the Ledger API. The participants jointly extend the ledger by running a distributed commit protocol.



The implementations might still rely on trusted third parties to facilitate the commit protocol. The required trust in terms of privacy and integrity, however, can generally be lower than in global state topologies. Moreover, unlike the previous topologies, they support interoperability: even if two transactions are committed with the help of disjoint sets of trusted third parties, their *output contracts* can in general still be used within the same atomic transaction. The exact trust assumptions and the degree of supported interoperability are implementation-dependent. Canton and DAML on R3 Corda are two such implementations. The main drawback of this topology is that availability can be influenced by the participant nodes. In particular, transactions cannot be committed if they use data that is only stored on unresponsive nodes. Spreading the data among additional trusted entities can mitigate the problem.

Chapter 5

SDK tools

5.1 DAML Assistant (`daml`)

`daml` is a command-line tool that does a lot of useful things related to the SDK. Using `daml`, you can:

- Create new DAML projects: `daml new <path to create project in>`
- Create a new project based on create-daml-app: `daml create-daml-app <path to create project in>`
- Initialize a DAML project: `daml init`
- Compile a DAML project: `daml build`

  This builds the DAML project according to the project config file `daml.yaml` (see *Configuration files* below).

  In particular, it will download and install the specified version of the 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.

5.1.1 Full help for commands

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

5.1.2 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

5.1.2.1 Global config file (daml-config.yaml)

The global config file daml-config.yaml is in the daml home directory (~/.daml on Linux and Mac, C:/Users/<user>/AppData/Roaming/daml on Windows). It controls options related to SDK version installation and upgrades.

By default it's blank, and you usually won't need to edit it. It recognizes the following options:

- auto-install: whether daml automatically installs a missing SDK version when it is required (defaults to true)
- update-check: how often daml will check for new versions of the SDK, in seconds (default to 86400, i.e. once a day)
  - This setting is only used to inform you when an update is available.
  - Set update-check: <number> to check for new versions every N seconds. Set update-check: never to never check for new versions.

Here is an example daml-config.yaml:

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

5.1.2.2 Project config file (daml.yaml)

The project config file daml.yaml must be in the root of your DAML project directory. It controls how the DAML project is built and how tools like Sandbox and Navigator interact with it.

The existence of a daml.yaml file is what tells daml that this directory contains a DAML project, and lets you use project-aware commands like daml build and daml start.

daml init creates a daml.yaml in an existing folder, so daml knows it's a project folder.

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

```yaml
sdk-version: __VERSION__
platform-version: __VERSION__
name: __PROJECT_NAME__
source: daml
scenario: Main:setup
parties:
  - Alice
  - Bob
version: 1.0.0
exposed-modules:
  - Main
dependencies:
  - daml-prim
  - daml-stdlib
```

(continues on next page)

(continued from previous page)

```
scenario-service:
  grpc-max-message-size: 134217728
  grpc-timeout: 60
  jvm-options: []
build-options: ["--ghc-option", "-Werror",
                "--ghc-option", "-v"]
```

Here is what each field means:

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

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

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

platform-version: Optional SDK version of platform components. Not setting this is equivalent to setting it to the same version as sdk-version. At the moment this includes Sandbox, Sandbox classic and the HTTP JSON API both when invoked directly via daml sandbox as well as when invoked via daml start. Changing the platform version is useful if you deploy to a ledger that is running on a different SDK version than you use locally and you want to make sure that you catch any issues during testing. E.g., you might compile your DAML code using SDK 1.3.0 so you get improvements in DAML Studio but deploy to DABL which could still be running a ledger and the JSON API from SDK 1.2.0. In that case, you can set sdk-version: 1.3.0 and platform-version: 1.2.0. It is possible to override the platform version by setting the DAML_PLATFORM_VERSION environment variable.

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

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

scenario: the name of the scenario to run when using daml start.

init-script: the name of the DAML script to run when using daml start.

parties: the parties to display in the Navigator when using daml start.

version: the project version.

exposed-modules: the DAML modules that are exposed by this project, which can be imported in other projects. If this field is not specified all modules in the project are exposed.

dependencies: library-dependencies of this project. See *Reference: DAML packages*.

data-dependencies: Cross-SDK dependencies of this project See *Reference: DAML packages*.

module-prefixes: Prefixes for all modules in package See *Reference: DAML packages*.

scenario-service: settings for the scenario service

- grpc-max-message-size: This option controls the maximum size of gRPC messages. If unspecified this defaults to 128MB (134217728 bytes). Unless you get errors, there should be no reason to modify this.
- grpc-timeout: This option controls the timeout used for communicating with the scenario service. If unspecified this defaults to 60s. Unless you get errors, there should be no reason to modify this.
- jvm-options: A list of options passed to the JVM when starting the scenario service. This can be used to limit maximum heap size via the -Xmx flag.

`build-options`: a list of tokens that will be appended to some invocations of `damlc` (currently *build* and *ide*). Note that there is no further shell parsing applied.
`sandbox-options`: a list of options that will be passed to Sandbox in `daml start`.
`navigator-options`: a list of options that will be passed to Navigator in `daml start`.
`json-api-options`: a list of options that will be passed to the HTTP JSON API in `daml start`.
`script-options`: a list of options that will be passed to the DAML script runner when running the `init-script` as part of `daml start`.
`start-navigator`: Controls whether navigator is started as part of `daml start`. Defaults to `true`. If this is specified as a CLI argument, say `daml start --start-navigator=true`, the CLI argument takes precedence over the value in `daml.yaml`.

5.1.3 Building DAML projects

To compile your DAML source code into a DAML archive (a `.dar` file), run:

```
daml build
```

You can control the build by changing your project's `daml.yaml`:

`sdk-version` The SDK version to use for building the project.
`name` The name of the project.
`source` The path to the source code.

The generated `.dar` file is created in `.daml/dist/${name}.dar` by default. To override the default location, pass the `-o` argument to `daml build`:

```
daml build -o path/to/darfile.dar
```

5.1.4 Managing SDK releases

You can manage SDK releases manually by using `daml install`.

To download and install the latest stable SDK release:

```
daml install latest
```

To download and install the latest snapshot release:

```
daml install latest --snapshots=yes
```

Please note that snapshot releases are not intended for production usage.

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

```
daml install project
```

To install a specific SDK version, for example version `0.13.55`, run:

```
daml install 0.13.55
```

Rarely, you might need to install an SDK release from a downloaded SDK release tarball. **This is an advanced feature**: you should only ever perform this on an SDK release tarball that is released through the official `digital-asset/daml` github repository. Otherwise your `daml` installation may become inconsistent with everyone else's. To do this, run:

```
daml install path-to-tarball.tar.gz
```

By default, `daml install` will update the assistant if the version being installed is newer. You can force the assistant to be updated with `--install-assistant=yes` and prevent the assistant from being updated with `--install-assistant=no`.

See `daml install --help` for a full list of options.

5.1.5 Terminal Command Completion

The `daml` assistant comes with support for `bash` and `zsh` completions. These will be installed automatically on Linux and Mac when you install or upgrade the DAML assistant.

If you use the `bash` shell, and your `bash` supports completions, you can use the TAB key to complete many `daml` commands, such as `daml install` and `daml version`.

For `Zsh` you first need to add `~/.daml/zsh` to your `$fpath`, e.g., by adding the following to the beginning of your `~/.zshrc` before you call `compinit`: `fpath=(~/.daml/zsh $fpath)`

You can override whether bash completions are installed for `daml` by passing `--bash-completions=yes` or `--bash-completions=no` to `daml install`.

5.2 DAML Studio

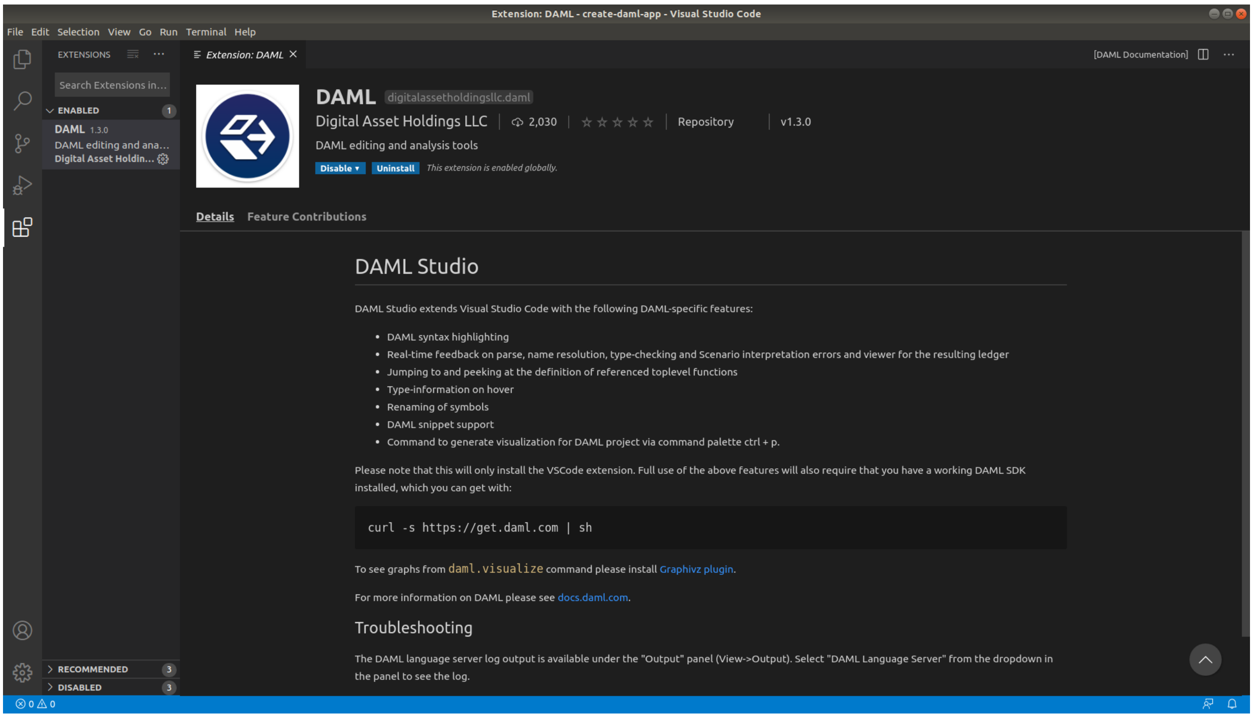
DAML Studio is an integrated development environment (IDE) for DAML. It is an extension on top of Visual Studio Code (VS Code), a cross-platform, open-source editor providing a rich code editing experience.

5.2.1 Installing

DAML Studio is included in the *DAML SDK*.

5.2.2 Creating your first DAML file

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



DAML Studio

DAML Studio extends Visual Studio Code with the following DAML-specific features:

- DAML syntax highlighting
- Real-time feedback on parse, name resolution, type-checking and Scenario interpretation errors and viewer for the resulting ledger
- Jumping to and peeking at the definition of referenced toplevel functions
- Type-information on hover
- Renaming of symbols
- DAML snippet support
- Command to generate visualization for DAML project via command palette ctrl + p.

Please note that this will only install the VSCode extension. Full use of the above features will also require that you have a working DAML SDK installed, which you can get with:

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

To see graphs from `daml.visualize` command please install Graphviz plugin.

For more information on DAML please see docs.daml.com.

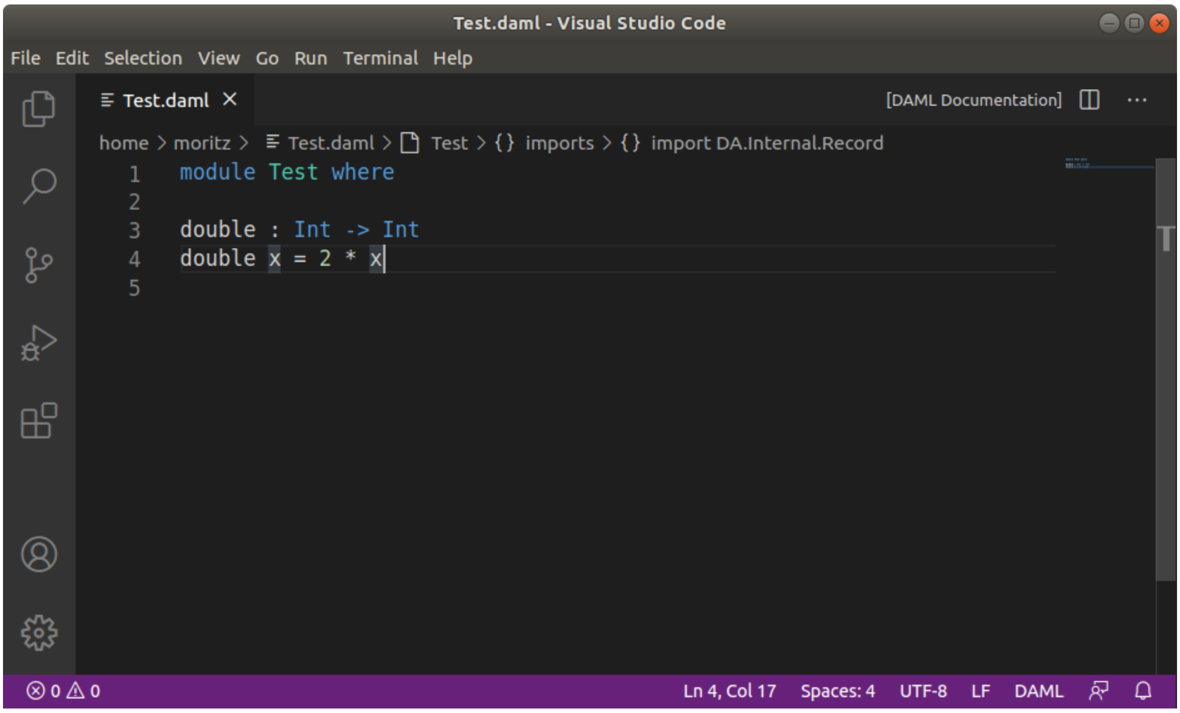
Troubleshooting

The DAML language server log output is available under the "Output" panel (View->Output). Select "DAML Language Server" from the dropdown in the panel to see the log.

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

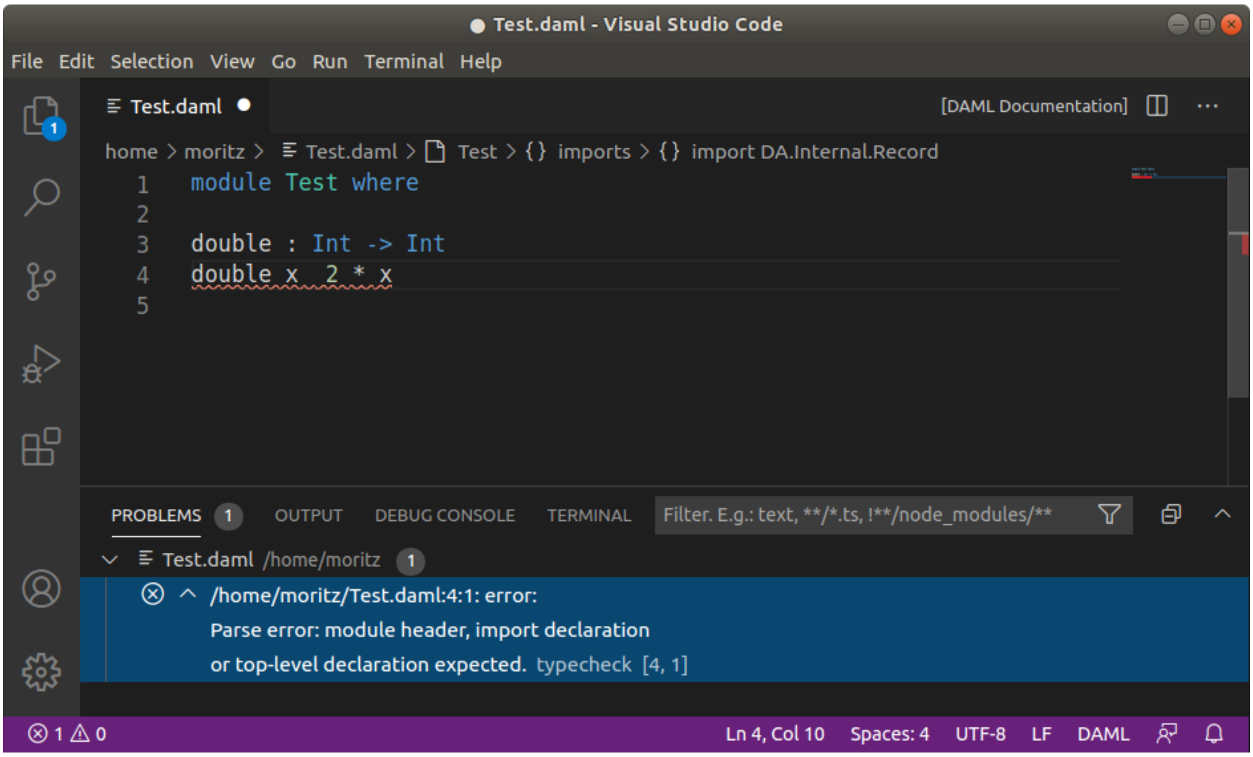
```
module Test where

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

Your screen should now look like the image below.


Test.daml - Visual Studio Code
File Edit Selection View Go Run Terminal Help
Test.daml
[DAML Documentation]
home > moritz > Test.daml > Test > {} imports > {} import DA.Internal.Record
⊗ 0 ⚠ 0
Ln 4, Col 17 Spaces: 4 UTF-8 LF DAML

```
module Test where

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

```

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.


● Test.daml - Visual Studio Code
File Edit Selection View Go Run Terminal Help
Test.daml ●
[DAML Documentation]
home > moritz > Test.daml > Test > {} imports > {} import DA.Internal.Record
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL
Filter. E.g.: text, **/*.ts, !**/node_modules/**
Test.daml /home/moritz 1
⊗ 1 ⚠ 0
Ln 4, Col 10 Spaces: 4 UTF-8 LF DAML

```
module Test where

double : Int -> Int
double x  2 * x

```

/home/moritz/Test.daml:4:1: error:
Parse error: module header, import declaration
or top-level declaration expected. typecheck [4, 1]

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*.

5.2.3 Supported features

Visual Studio Code provides many helpful features for editing DAML files and we recommend reviewing Visual Studio Code Basics and Visual Studio Code Keyboard Shortcuts for OS X. The DAML Studio extension for Visual Studio Code provides the following DAML-specific features:

5.2.3.1 Symbols and problem reporting

Use the commands listed below to navigate between symbols, rename them, and inspect any problems detected in your DAML files. Symbols are identifiers such as template names, lambda arguments, variables, and so on.

Command	Shortcut (OS X)
Go to Definition	F12
Peek Definition	□F12
Rename Symbol	F2
Go to Symbol in File	□□O
Go to Symbol in Workspace	□T
Find all References	□F12
Problems Panel	□□M

Note: You can also start a command by typing its name into the command palette (press □□P or F1). The command palette is also handy for looking up keyboard shortcuts.

Note:

Rename Symbol, Go to Symbol in File, Go to Symbol in Workspace, and Find all References work on: choices, record fields, top-level definitions, let-bound variables, lambda arguments, and modules

Go to Definition and Peek Definition work on: top-level definitions, let-bound variables, lambda arguments, and modules

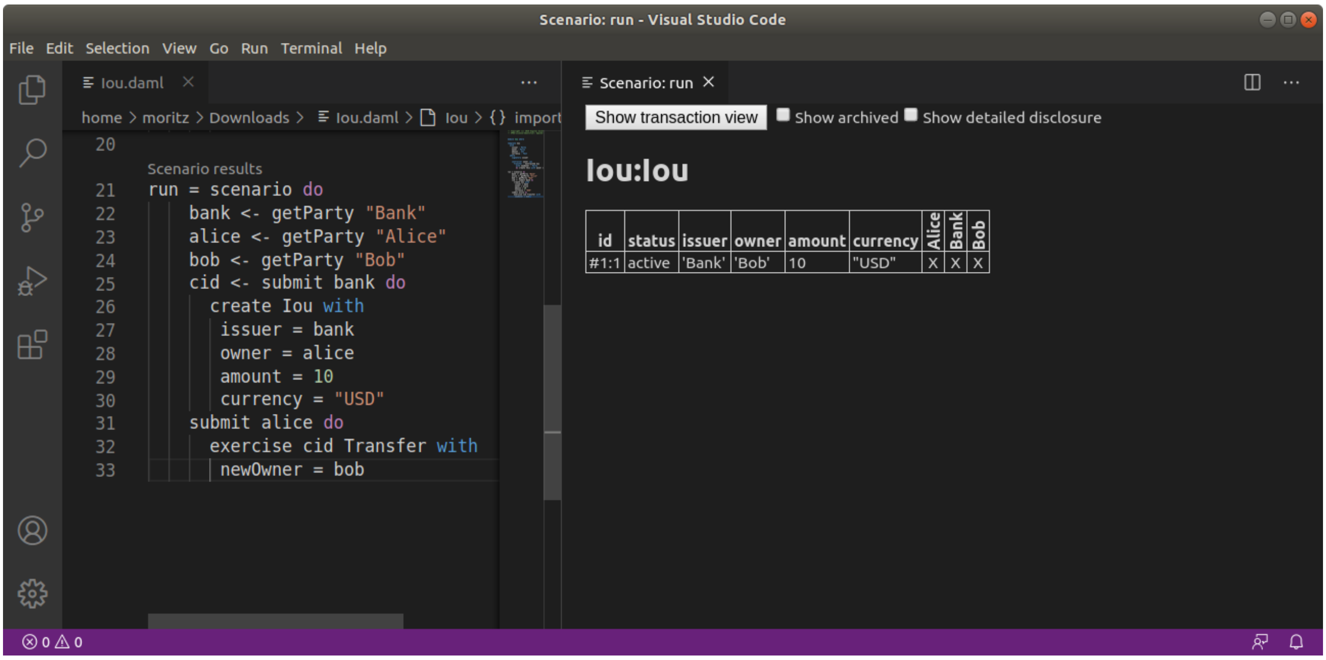
5.2.3.2 Hover tooltips

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

5.2.3.3 Scenario and DAML Script results

Top-level declarations of type `Scenario` or `Script` are decorated with a `Scenario results` or a `Script results` code lens. You can click on the code lens to inspect the execution transaction graph and the active contracts. The functionality for inspecting the results is identical for DAML Scripts and scenarios.

For the scenario from the `Iou` module, you get the following table displaying all contracts that are active at the end of the scenario. The first column displays the contract id. The columns afterwards represent the fields of the contract and finally you get one column per party with an `X` if the party can see the contract or a – if not.


Scenario: run - Visual Studio Code
File Edit Selection View Go Run Terminal Help
Iou.daml
home > moritz > Downloads > Iou.daml > Iou > {} import
Scenario: run
Show transaction view
□ Show archived
□ Show detailed disclosure
Scenario results
⊗ 0 ⚠ 0

```
run = scenario do
    bank <- getParty "Bank"
    alice <- getParty "Alice"
    bob <- getParty "Bob"
    cid <- submit bank do
      create Iou with
       issuer = bank
       owner = alice
       amount = 10
       currency = "USD"
    submit alice do
      exercise cid Transfer with
       newOwner = bob
```

Iou:Iou

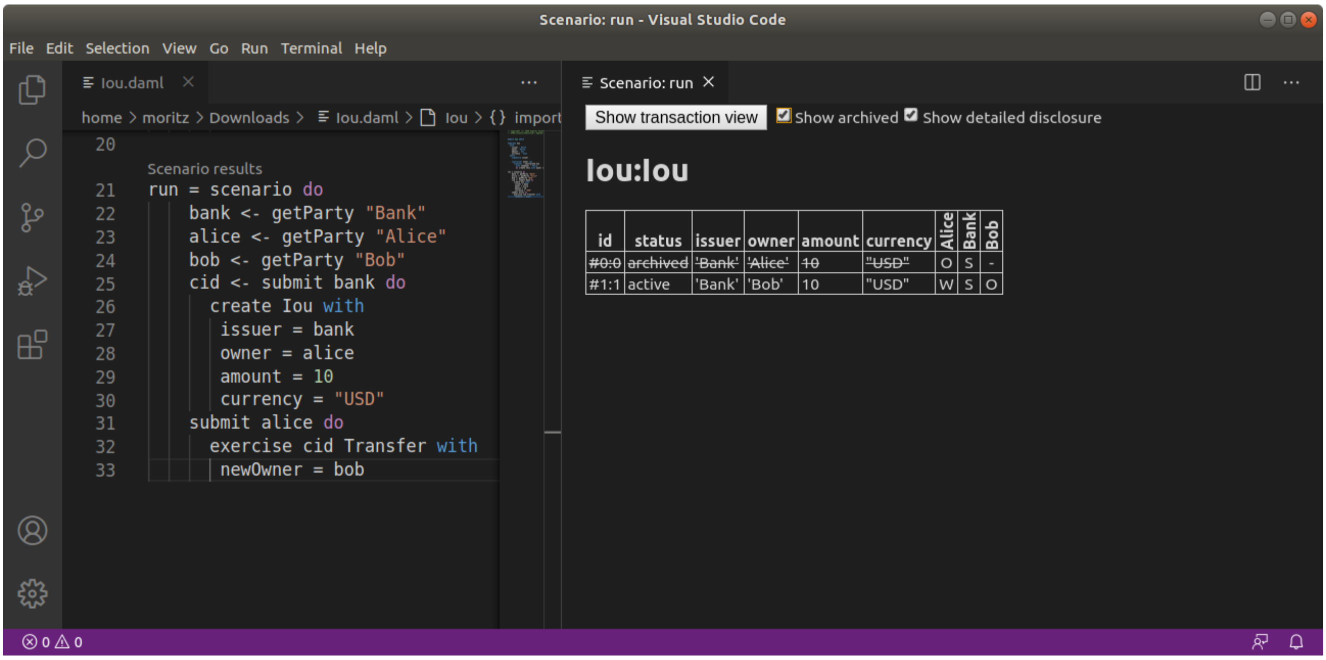
id	status	issuer	owner	amount	currency	Alice	Bank	Bob
#1:1	active	'Bank'	'Bob'	10	"USD"	X	X	X

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

1. `S`, the party sees the contract because they are a signatory on the contract.
2. `O`, the party sees the contract because they are an observer on the contract.
3. `W`, the party sees the contract because they witnessed the creation of this contract, e.g., because they are an actor on the `exercise` that created it.

4. D, the party sees the contract because they have been divulged the contract, e.g., because they witnessed an exercise that resulted in a fetch of this contract.

For details on the meaning of those four categories, refer to the *DAML Ledger Model*. For the example above, the resulting table looks as follows. You can see the archived Bank contract and the active Bank contract whose creation Alice has witnessed by virtue of being an actor on the exercise that created it.


Scenario: run - Visual Studio Code
File Edit Selection View Go Run Terminal Help
Iou.daml
home > moritz > Downloads > Iou.daml > Iou > {} import
Scenario: run
Show transaction view
☑ Show archived ☑ Show detailed disclosure
⊗ 0 ⚠ 0

```
20
   Scenario results
21 run = scenario do
22     bank <- getParty "Bank"
23     alice <- getParty "Alice"
24     bob <- getParty "Bob"
25     cid <- submit bank do
26       create Iou with
27         issuer = bank
28         owner = alice
29         amount = 10
30         currency = "USD"
31     submit alice do
32       exercise cid Transfer with
33         newOwner = bob
```

Iou:Iou

id	status	issuer	owner	amount	currency	Alice	Bank	Bob
~~#0:0~~	~~archived~~	~~'Bank'~~	~~'Alice'~~	~~10~~	~~"USD"~~	O	S	-
#1:1	active	'Bank'	'Bob'	10	"USD"	W	S	O

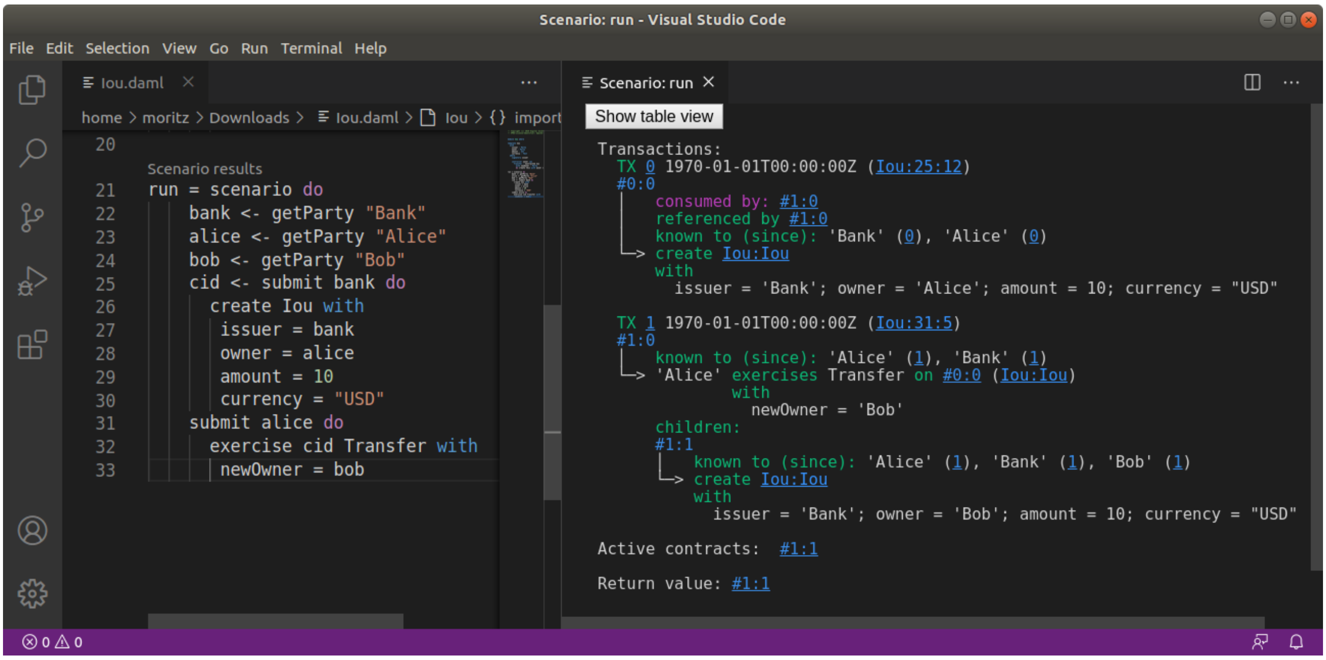
If you want to see the detailed transaction graph you can click on the Show transaction view button. The transaction graph consists of transactions, each of which contain one or more updates to the ledger, that is creates and exercises. The transaction graph also records fetches of contracts.

For example a scenario for the Iou module looks as follows:

Each transaction is the result of executing a step in the scenario. In the image below, the transaction #0 is the result of executing the first line of the scenario (line 20), where the Iou is created by the bank. The following information can be gathered from the transaction:

> The result of the first scenario transaction #0 was the creation of the Iou contract with the arguments bank, 10, and "USD".
> The created contract is referenced in transaction #1, step 0.
> The created contract was consumed in transaction #1, step 0.
> A new contract was created in transaction #1, step 1, and has been divulged to parties 'Alice', 'Bob', and 'Bank'.
> At the end of the scenario only the contract created in #1:1 remains.
> The return value from running the scenario is the contract identifier #1:1.
> And finally, the contract identifiers assigned in scenario execution correspond to the scenario step that created them (e.g. #1).

You can navigate to the corresponding source code by clicking on the location shown in parenthesis (e.g. Iou:25:12, which means the Iou module, line 25 and column 1). You can also navigate between transactions by clicking on the transaction and contract ids (e.g. #1:0).


Scenario: run - Visual Studio Code
File Edit Selection View Go Run Terminal Help
Iou.daml
home > moritz > Downloads > Iou.daml > Iou > {} import
Scenario: run
Show table view
Scenario results
⊗ 0 ⚠ 0

```
run = scenario do
    bank <- getParty "Bank"
    alice <- getParty "Alice"
    bob <- getParty "Bob"
    cid <- submit bank do
      create Iou with
        issuer = bank
        owner = alice
        amount = 10
        currency = "USD"
    submit alice do
      exercise cid Transfer with
        newOwner = bob
```

```
Transactions:
  TX 0 1970-01-01T00:00:00Z (Iou:25:12)
  #0:0
  │   consumed by: #1:0
  │   referenced by #1:0
  │   known to (since): 'Bank' (0), 'Alice' (0)
  └─> create Iou:Iou
      with
        issuer = 'Bank'; owner = 'Alice'; amount = 10; currency = "USD"

  TX 1 1970-01-01T00:00:00Z (Iou:31:5)
  #1:0
  │   known to (since): 'Alice' (1), 'Bank' (1)
  └─> 'Alice' exercises Transfer on #0:0 (Iou:Iou)
              with
                newOwner = 'Bob'
      children:
      #1:1
      │   known to (since): 'Alice' (1), 'Bank' (1), 'Bob' (1)
      └─> create Iou:Iou
          with
            issuer = 'Bank'; owner = 'Bob'; amount = 10; currency = "USD"

Active contracts:  #1:1

Return value: #1:1
```

Fig. 1: Scenario results

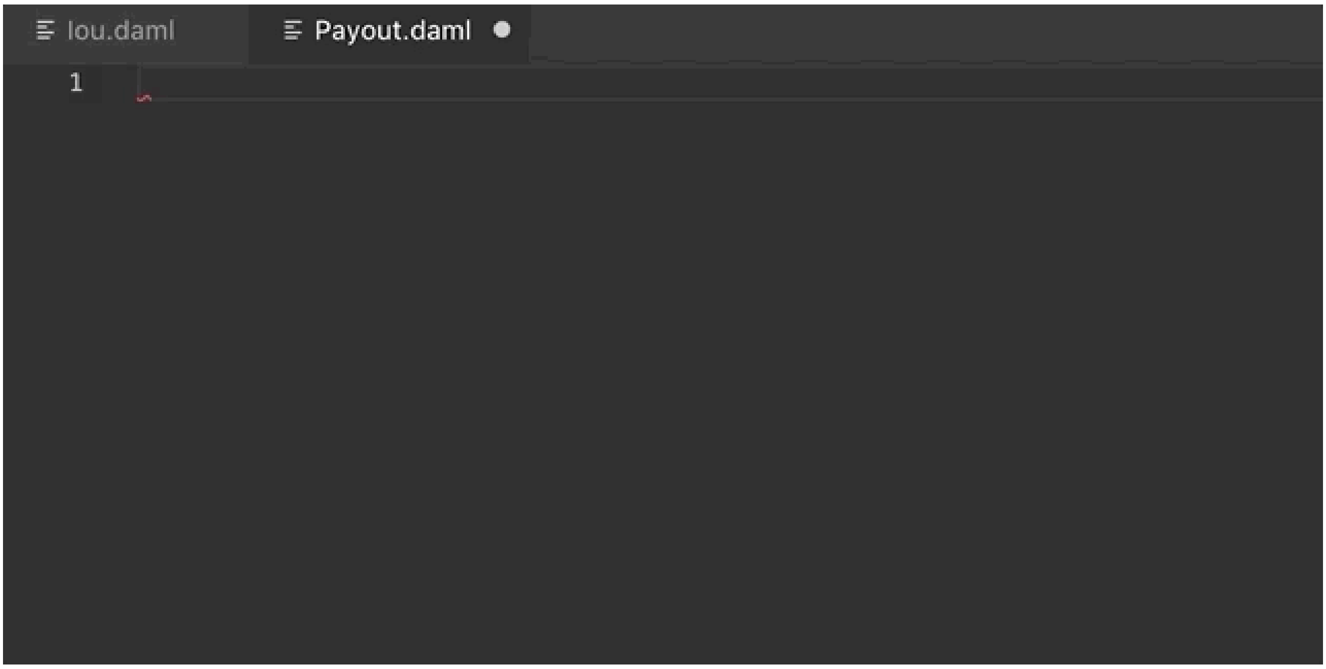
5.2.3.4 DAML snippets

You can automatically complete a number of snippets when editing a DAML source file. By default, hitting `^-Space` after typing a DAML keyword displays available snippets that you can insert.

To define your own workflow around DAML snippets, adjust your user settings in Visual Studio Code to include the following options:

```
{
  "editor.tabCompletion": true,
  "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.

5.2.4 Common scenario errors

During DAML execution, errors can occur due to exceptions (e.g. use of abort , or division by zero), or due to authorization failures. You can expect to run into the following errors when writing DAML.

When a runtime error occurs in a scenario execution, the scenario result view shows the error together with the following additional information, if available:

Location of the failed commit If the failing part of the script was a `submit`, the source location of the call to `submit` will be displayed.

Stack trace A list of source locations that were encoutered before the error occured. The last encountered location is the first entry in the list.

Ledger time The ledger time at which the error occurred.

Partial transaction The transaction that is being constructed, but not yet committed to the ledger.

Committed transaction Transactions that were successfully committed to the ledger prior to the error.

Trace Any messages produced by calls to `trace` and `debug`.

5.2.4.1 Abort, assert, and debug

The `abort`, `assert` and `debug` inbuilt functions can be used in updates and scenarios. All three can be used to output messages, but `abort` and `assert` can additionally halt the execution:

```
abortTest = scenario do
  debug "hello, world!"
  abort "stop"
```

```
Scenario execution failed:
  Aborted:  stop

```

(continues on next page)

(continued from previous page)

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

Partial transaction:

Trace:
  "hello, world!"
```

5.2.4.2 Missing authorization on create

If a contract is being created without approval from all authorizing parties the commit will fail. For example:

```
template Example
  with
    party1 : Party; party2 : Party
  where
    signatory party1
    signatory party2

example = scenario do
  alice <- getParty "Alice"
  bob <- getParty "Bob"
  submit alice (create Example with party1=alice; party2=bob)
```

Execution of the example scenario fails due to 'Bob' being a signatory in the contract, but not authorizing the create:

```
Scenario execution failed:
  #0: create of CreateAuthFailure:Example at unknown source
      failed due to a missing authorization from 'Bob'

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

Partial transaction:
  Sub-transactions:
     #0
     └─> create CreateAuthFailure:Example
         with
           party1 = 'Alice'; party2 = 'Bob'
```

To create the Example contract one would need to bring both parties to authorize the creation via a choice, for example 'Alice' could create a contract giving 'Bob' the choice to create the 'Example' contract.

5.2.4.3 Missing authorization on exercise

Similarly to creates, exercises can also fail due to missing authorizations when a party that is not a controller of a choice exercises it.

```
template Example
  with
    owner : Party
    friend : Party
  where
    signatory owner

    controller owner can
      Consume : ()
        do return ()

    controller friend can
      Hello : ()
        do return ()

example = scenario do
  alice <- getParty "Alice"
  bob <- getParty "Bob"
  cid <- submit alice (create Example with owner=alice; friend=bob)
  submit bob do exercise cid Consume
```

The execution of the example scenario fails when 'Bob' tries to exercise the choice 'Consume' of which he is not a controller

```
Scenario execution failed:
  #1: exercise of Consume in ExerciseAuthFailure:Example at unknown source
      failed due to a missing authorization from 'Alice'

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

Partial transaction:
  Sub-transactions:
     #0
     └─> fetch #0:0 (ExerciseAuthFailure:Example)

     #1
     └─> 'Alice' exercises Consume on #0:0 (ExerciseAuthFailure:Example)
                 with

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

From the error we can see that the parties authorizing the exercise ('Bob') is not a subset of the required controlling parties.

5.2.4.4 Contract not visible

Contract not being visible is another common error that can occur when a contract that is being fetched or exercised has not been disclosed to the committing party. For example:

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

    controller owner can
      Consume : ()
        do return ()

example = scenario do
  alice <- getParty "Alice"
  bob <- getParty "Bob"
  cid <- submit alice (create Example with owner=alice)
  submit bob do exercise cid Consume
```

In the above scenario the 'Example' contract is created by 'Alice' and makes no mention of the party 'Bob' and hence does not cause the contract to be disclosed to 'Bob'. When 'Bob' tries to exercise the contract the following error would occur:

```
Scenario execution failed:
  Attempt to fetch or exercise a contract not visible to the committer.
  Contract:  #0:0 (NotVisibleFailure:Example)
  Committer: 'Bob'
  Disclosed to: 'Alice'

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

Partial transaction:

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

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

5.2.5 Working with multiple packages

Often a DAML project consists of multiple packages, e.g., one containing your templates and one containing a DAML trigger so that you can keep the templates stable while modifying the trigger. It is possible to work on multiple packages in a single session of DAML studio but you have to keep some things in mind. You can see the directory structure of a simple multi-package project consisting of two packages `pkga` and `pkgb` below:

```
.
├── daml.yaml
├── pkga
│   ├── daml
│   │   └── A.daml
│   └── daml.yaml
└── pkgb
    ├── daml
    │   └── B.daml
    └── daml.yaml
```

pkga and pkgb are regular DAML projects with a daml.yaml and a DAML module. In addition to the daml.yaml files for the respective packages, you also need to add a daml.yaml to the root of your project. This file only needs to specify the SDK version. Replace X.Y.Z by the SDK version you specified in the daml.yaml files of the individual packages. Note that this feature is only available in SDK version 0.13.52 and newer.

```
sdk-version: X.Y.Z
```

You can then open DAML Studio once in the root of your project and work on files in both packages. Note that if pkgb refers to pkga.dar in its dependencies field, changes will not be picked up automatically. This is always the case even if you open DAML Studio in pkgb. However, for multi-package projects there is an additional caveat: You have to both rebuild pkga.dar using daml build and then build pkgb using daml build before restarting DAML Studio.

5.3 DAML Sandbox

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

You can start Sandbox together with *Navigator* using the daml start command in a DAML 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.

Note: Sandbox has switched to use Wall Clock Time mode by default. To use Static Time Mode you can provide the --static-time flag to the daml sandbox command or configure the time mode for daml start in sandbox-options: section of daml.yaml. Please refer to *DAML configuration files* for more information.

Sandbox can also be run manually as in this example:

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


   ____             ____
  / __/__ ____  ___/ / /  ___ __ __
 _\ \/ _ `/ _ \/ _  / _ \/ _ \\ \ /
/___/\_,_/_//_/\_,_/_.__/\___/_\_\
initialized sandbox with ledger-id = sandbox-16ae201c-b2fd-45e0-af04-
↪c61abe13fed7, port = 6865,
```

(continues on next page)

(continued from previous page)

```
dar file = DAR files at List(/Users/damluser/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 `:`. We also specify that the Sandbox should run in Static Time mode so that the scenario can control the time.

Note: The scenario is used for testing and development only, and is not supported by production DAML Ledgers. It is therefore inadvisable 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.

5.3.1 Contract Identifier Generation

Sandbox supports two contract identifier generator schemes:

> The so-called *deterministic* scheme that deterministically produces contract identifiers from the state of the underlying ledger. Those identifiers are strings starting with `#`.
> The so-called *random* scheme that produces contract identifiers indistinguishable from random. In practice, the schemes use a cryptographically secure pseudorandom number generator initialized with a truly random seed. Those identifiers are hexadecimal strings prefixed by `00`.

The sandbox can be configured to use one or the other scheme with one of the following command line options:

> `--contract-id-seeding=<seeding-mode>`. The Sandbox will use the seeding mode *<seeding-mode>* to seed the generation of random contract identifiers. Possible seeding modes are:
>
> - `no`: The Sandbox uses the `deterministic` scheme.
> - `strong`: The Sandbox uses the `random` scheme initialized with a high-entropy seed. Depending on the underlying operating system, the startup of the Sandbox may block as entropy is being gathered to generate the seed.
> - `testing-weak`: (**For testing purposes only**) The Sandbox uses the `random` scheme initialized with a low entropy seed. This may be used in a testing environment to avoid exhausting the system entropy pool when a large number of Sandboxes are started in a short time interval.
> - `testing-static`: (**For testing purposes only**) The sandbox uses the `random` scheme with a fixed seed. This may be used in testing for reproducible runs.

5.3.2 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

5.3.3 Running with authentication

By default, Sandbox does not use any authentication and accepts all valid ledger API requests.

To start Sandbox with authentication based on JWT tokens, use one of the following command line options:

- `--auth-jwt-rs256-crt=<filename>`. The sandbox will expect all tokens to be signed with RS256 (RSA Signature with SHA-256) with the public key loaded from the given X.509 certificate file. Both PEM-encoded certificates (text files starting with `-----BEGIN CERTIFICATE-----`) and DER-encoded certificates (binary files) are supported.
- `--auth-jwt-es256-crt=<filename>`. The sandbox will expect all tokens to be signed with ES256 (ECDSA using P-256 and SHA-256) with the public key loaded from the given X.509 certificate file. Both PEM-encoded certificates (text files starting with `-----BEGIN CERTIFICATE-----`) and DER-encoded certicates (binary files) are supported.
- `--auth-jwt-es512-crt=<filename>`. The sandbox will expect all tokens to be signed with ES512 (ECDSA using P-521 and SHA-512) with the public key loaded from the given X.509 certificate file. Both PEM-encoded certificates (text files starting with `-----BEGIN CERTIFICATE-----`) and DER-encoded certificates (binary files) are supported.
- `--auth-jwt-rs256-jwks=<url>`. The sandbox will expect all tokens to be signed with RS256 (RSA Signature with SHA-256) with the public key loaded from the given JWKS URL.

Warning: For testing purposes only, the following options may also be used. None of them is considered safe for production:

- `--auth-jwt-hs256-unsafe=<secret>`. The sandbox will expect all tokens to be signed with HMAC256 with the given plaintext secret.

5.3.3.1 Token payload

JWTs express claims which are documented in the authorization documentation.

The following is an example of a valid JWT payload:

```
{
   "https://daml.com/ledger-api": {
     "ledgerId": "aaaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee",
     "participantId": null,
     "applicationId": null,
     "admin": true,
     "actAs": ["Alice"],
     "readAs": ["Bob"]
   },
   "exp": 1300819380
}
```

where

ledgerId, participantId, applicationId restricts the validity of the token to the given ledger, participant, or application
exp is the standard JWT expiration date (in seconds since EPOCH)
admin, actAs and readAs bear the same meaning as in the authorization documentation

The public claim is implicitly held by anyone bearing a valid JWT (even without being an admin or being able to act or read on behalf of any party).

5.3.3.2 Generating JSON Web Tokens (JWT)

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

5.3.3.3 Generating RSA keys

To generate RSA keys for testing purposes, use the following command

```
openssl req -nodes -new -x509 -keyout sandbox.key -out sandbox.crt
```

which generates the following files:

sandbox.key: the private key in PEM/DER/PKCS#1 format
sandbox.crt: a self-signed certificate containing the public key, in PEM/DER/X.509 Certificate format

5.3.3.4 Generating EC keys

To generate keys to be used with ES256 for testing purposes, use the following command

```
openssl req -x509 -nodes -days 3650 -newkey ec:<(openssl ecparam -name
↪prime256v1) -keyout ecdsa256.key -out ecdsa256.crt
```

which generates the following files:

ecdsa256.key: the private key in PEM/DER/PKCS#1 format
ecdsa256.crt: a self-signed certificate containing the public key, in PEM/DER/X.509 Certificate format

Similarly, you can use the following command for ES512 keys:

```
openssl req -x509 -nodes -days 3650 -newkey ec:<(openssl ecparam -name
↪secp521r1) -keyout ecdsa512.key -out ecdsa512.crt
```

5.3.4 Running with TLS

To enable TLS, you need to specify the private key for your server and the certificate chain via `daml sandbox --pem server.pem --crt server.crt`. By default, Sandbox requires client authentication as well. You can set a custom root CA certificate used to validate client certificates via `--cacrt ca.crt`. You can change the client authentication mode via `--client-auth none` which will disable it completely, `--client-auth optional` which makes it optional or specify the default explicitly via `-.client-auth require`.

5.3.5 Command-line reference

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

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

5.3.6 Metrics

5.3.6.1 Enable and configure reporting

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

> `--metrics-reporter`: passing a legal value will enable reporting; the accepted values are `console`, `csv:</path/to/metrics.csv>` and `graphite:<local_server_port>`.
> - `console`: prints captured metrics on the standard output
> - `csv://</path/to/metrics.csv>`: saves the captured metrics in CSV format at the specified location
> - `graphite://<server_host>[:<server_port>]`: sends captured metrics to a Graphite server. If the port is omitted, the default value `2003` will be used.
>
> `--metrics-reporting-interval`: metrics are pre-aggregated on the sandbox and sent to the reporter, this option allows the user to set the interval. The formats accepted are based on the ISO-8601 duration format `PnDTnHnMn.nS` with days considered to be exactly 24 hours. The default interval is 10 seconds.

5.3.6.2 Types of metrics

This is a list of type of metrics with all data points recorded for each. Use this as a reference when reading the list of metrics.

Gauge

An individual instantaneous measurement.

Counter

Number of occurrences of some event.

Meter

A meter tracks the number of times a given event occurred. The following data points are kept and reported by any meter.

> `<metric.qualified.name>.count`: number of registered data points overall
> `<metric.qualified.name>.m1_rate`: number of registered data points per minute
> `<metric.qualified.name>.m5_rate`: number of registered data points every 5 minutes
> `<metric.qualified.name>.m15_rate`: number of registered data points every 15 minutes

`<metric.qualified.name>.mean_rate`: mean number of registered data points

Histogram

An histogram records aggregated statistics about collections of events. The exact meaning of the number depends on the metric (e.g. timers are histograms about the time necessary to complete an operation).

`<metric.qualified.name>.mean`: arithmetic mean
`<metric.qualified.name>.stddev`: standard deviation
`<metric.qualified.name>.p50`: median
`<metric.qualified.name>.p75`: 75th percentile
`<metric.qualified.name>.p95`: 95th percentile
`<metric.qualified.name>.p98`: 98th percentile
`<metric.qualified.name>.p99`: 99th percentile
`<metric.qualified.name>.p999`: 99.9th percentile
`<metric.qualified.name>.min`: lowest registered value overall
`<metric.qualified.name>.max`: highest registered value overall

Histograms only keep a small *reservoir* of statistically relevant data points to ensure that metrics collection can be reasonably accurate without being too taxing resource-wise.

Unless mentioned otherwise all histograms (including timers, mentioned below) use exponentially decaying reservoirs (i.e. the data is roughly relevant for the last five minutes of recording) to ensure that recent and possibly operationally relevant changes are visible through the metrics reporter.

Note that `min` and `max` values are not affected by the reservoir sampling policy.

You can read more about reservoir sampling and possible associated policies in the Dropwizard Metrics library documentation.

Timers

A timer records all metrics registered by a meter and by an histogram, where the histogram records the time necessary to execute a given operation (unless otherwise specified, the precision is nanoseconds and the unit of measurement is milliseconds).

Database Metrics

A database metric is a collection of simpler metrics that keep track of relevant numbers when interacting with a persistent relational store.

These metrics are:

`<metric.qualified.name>.wait` (timer): time to acquire a connection to the database
`<metric.qualified.name>.exec` (timer): time to run the query and read the result
`<metric.qualified.name>.query` (timer): time to run the query
`<metric.qualified.name>.commit` (timer): time to perform the commit
`<metric.qualified.name>.translation` (timer): if relevant, time necessary to turn serialized DAML-LF values into in-memory objects

5.3.6.3 List of metrics

The following is a non-exhaustive list of selected metrics that can be particularly important to track. Note that not all the following metrics are available unless you run the sandbox with a PostgreSQL backend.

`daml.commands.deduplicated_commands`

A meter. Number of deduplicated commands.

`daml.commands.delayed_submissions`

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

`daml.commands.failed_command_interpretation`

A meter. Number of commands that have been deemed unacceptable by the interpreter and thus rejected (e.g. double spends)

`daml.commands.submissions`

A timer. Time to fully process a submission (validation, deduplication and interpretation) before it's handed over to the ledger to be finalized (either committed or rejected).

`daml.commands.valid_submissions`

A meter. Number of submission that pass validation and are further sent to deduplication and interpretation.

`daml.commands.validation`

A timer. Time to validate submitted commands before they are fed to the DAML interpreter.

`daml.execution.get_lf_package`

A timer. Time spent by the engine fetching the packages of compiled DAML code necessary for interpretation.

`daml.execution.lookup_active_contract_count_per_execution`

A histogram. Number of active contracts fetched for each processed transaction.

`daml.execution.lookup_active_contract_per_execution`

A timer. Time to fetch all active contracts necessary to process each transaction.

`daml.execution.lookup_active_contract`

A timer. Time to fetch each individual active contract during interpretation.

`daml.execution.lookup_contract_key_count_per_execution`

A histogram. Number of contract keys looked up for each processed transaction.

`daml.execution.lookup_contract_key_per_execution`

A timer. Time to lookup all contract keys necessary to process each transaction.

`daml.execution.lookup_contract_key`

A timer. Time to lookup each individual contract key during interpretation.

`daml.execution.retry`

A meter. Overall number of interpretation retries attempted due to mismatching ledger effective time.

`daml.execution.total`

A timer. Time spent interpreting a valid command into a transaction ready to be submitted to the ledger for finalization.

`daml.index.db.connection.sandbox.pool`

This namespace holds a number of interesting metrics about the connection pool used to communicate with the persistent store that underlies the index.

These metrics include:

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

`daml.index.db.deduplicate_command`

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

`daml.index.db.get_active_contracts`

A database metric. Time spent retrieving a page of active contracts to be served from the active contract service. The page size is configurable, please look at the CLI reference.

`daml.index.db.get_completions`

A database metric. Time spent retrieving a page of command completions to be served from the command completion service. The page size is configurable, please look at the CLI reference.

`daml.index.db.get_flat_transactions`

A database metric. Time spent retrieving a page of flat transactions to be streamed from the transaction service. The page size is configurable, please look at the CLI reference.

`daml.index.db.get_ledger_end`

A database metric. Time spent retrieving the current ledger end. The count for this metric is expected to be very high and always increasing as the indexed is queried for the latest updates.

`daml.index.db.get_ledger_id`

A database metric. Time spent retrieving the ledger identifier.

`daml.index.db.get_transaction_trees`

A database metric. Time spent retrieving a page of flat transactions to be streamed from the transaction service. The page size is configurable, please look at the CLI reference.

`daml.index.db.load_all_parties`

A database metric. Load the currently allocated parties so that they are served via the party management service.

`daml.index.db.load_archive`

A database metric. Time spent loading a package of compiled DAML code so that it's given to the DAML interpreter when needed.

`daml.index.db.load_configuration_entries`

A database metric. Time to load the current entries in the log of configuration entries. Used to verify whether a configuration has been ultimately set.

`daml.index.db.load_package_entries`

A database metric. Time to load the current entries in the log of package uploads. Used to verify whether a package has been ultimately uploaded.

`daml.index.db.load_packages`

A database metric. Load the currently uploaded packages so that they are served via the package management service.

`daml.index.db.load_parties`

A database metric. Load the currently allocated parties so that they are served via the party service.

`daml.index.db.load_party_entries`

A database metric. Time to load the current entries in the log of party allocations. Used to verify whether a party has been ultimately allocated.

`daml.index.db.lookup_active_contract`

A database metric. Time to fetch one contract on the index to be used by the DAML interpreter to evaluate a command into a transaction.

`daml.index.db.lookup_configuration`

A database metric. Time to fetch the configuration so that it's served via the configuration management service.

`daml.index.db.lookup_contract_by_key`

A database metric. Time to lookup one contract key on the index to be used by the DAML interpreter to evaluate a command into a transaction.

`daml.index.db.lookup_flat_transaction_by_id`

A database metric. Time to lookup a single flat transaction by identifier to be served by the transaction service.

`daml.index.db.lookup_maximum_ledger_time`

A database metric. Time spent looking up the ledger effective time of a transaction as the maximum ledger time of all active contracts involved to ensure causal monotonicity.

`daml.index.db.lookup_transaction_tree_by_id`

A database metric. Time to lookup a single transaction tree by identifier to be served by the transaction service.

`daml.index.db.remove_expired_deduplication_data`

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

`daml.index.db.stop_deduplicating_command`

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

`daml.index.db.store_configuration_entry`

A database metric. Time spent persisting a change in the ledger configuration provided through the configuration management service.

`daml.index.db.store_ledger_entry`

A database metric. Time spent persisting a transaction that has been successfully interpreted and is final.

`daml.index.db.store_package_entry`

A database metric. Time spent storing a DAML package uploaded through the package management service.

`daml.index.db.store_party_entry`

A database metric. Time spent storing party information as part of the party allocation endpoint provided by the party management service.

`daml.index.db.store_rejection`

A database metric. Time spent persisting the information that a given command has been rejected.

`daml.lapi`

Every metrics under this namespace is a timer, one for each service exposed by the Ledger API, in the format:

`daml.lapi.service_name.service_endpoint`

As in the following example:

`daml.lapi.command_service.submit_and_wait`

Single call services return the time to serve the request, streaming services measure the time to return the first response.

`jvm`

Under the `jvm` namespace there is a collection of metrics that tracks important measurements about the JVM that the sandbox is running on, including CPU usage, memory consumption and the current state of threads.

5.4 Navigator

The Navigator is a front-end that you can use to connect to any DAML Ledger and inspect and modify the ledger. You can use it during DAML development to explore the flow and implications of the DAML models.

The first sections of this guide cover use of the Navigator with the DAML SDK. Refer to *Advanced usage* for information on using Navigator outside the context of the SDK.

5.4.1 Navigator functionality

Connect Navigator to any DAML Ledger and use it to:

- View templates
- View active and archived contracts
- Exercise choices on contracts
- Advance time (This option applies only when using Navigator with the DAML Sandbox ledger.)

5.4.2 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

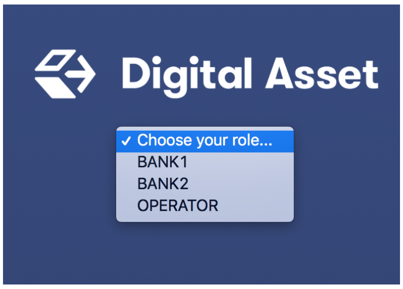
When running `daml start` you will see the Navigator URL. By default it will be http://localhost:7500/.

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.

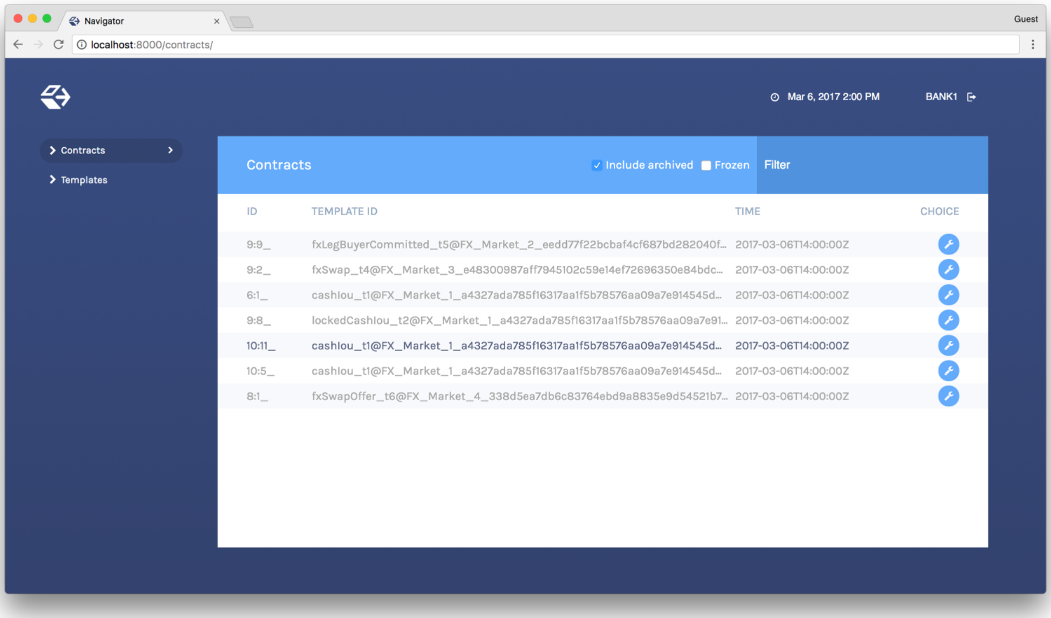
5.4.3 Choosing a party / changing the party

The ledger is a record of transactions between authorized participants on the distributed network. Before you can interact with the ledger, you must assume the role of a particular party. This determines the contracts that you can access and the actions you are permitted to perform on the ledger. The first step in using Navigator is to use the drop-down list on the Navigator home screen to select from the available parties.



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

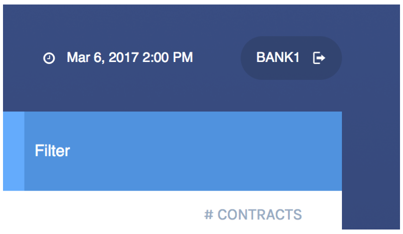
The main Navigator screen will be displayed, with contracts that this party is entitled to view in the main pane and the option to switch from contracts to templates in the pane at the left. Other options allow you to filter the display, include or exclude archived contracts, and exercise choices as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



ID	TEMPLATE ID	TIME	CHOICE
9:9_	fxLegBuyerCommitted_t5@FX_Market_2_eedd77f22bcbaf4cf687bd282040f...	2017-03-06T14:00:00Z	
9:2_	fxSwap_t4@FX_Market_3_e48300987aff7945102c59e14ef72696350e84bdc...	2017-03-06T14:00:00Z	
6:1_	cashIou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d...	2017-03-06T14:00:00Z	
9:8_	lockedCashIou_t2@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e91...	2017-03-06T14:00:00Z	
10:11_	cashIou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d...	2017-03-06T14:00:00Z	
10:5_	cashIou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d...	2017-03-06T14:00:00Z	
8:1_	fxSwapOffer_t6@FX_Market_4_338d5ea7db6c83764ebd9a8835e9d54521b7...	2017-03-06T14:00:00Z	

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.

5.4.4 Logging out

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

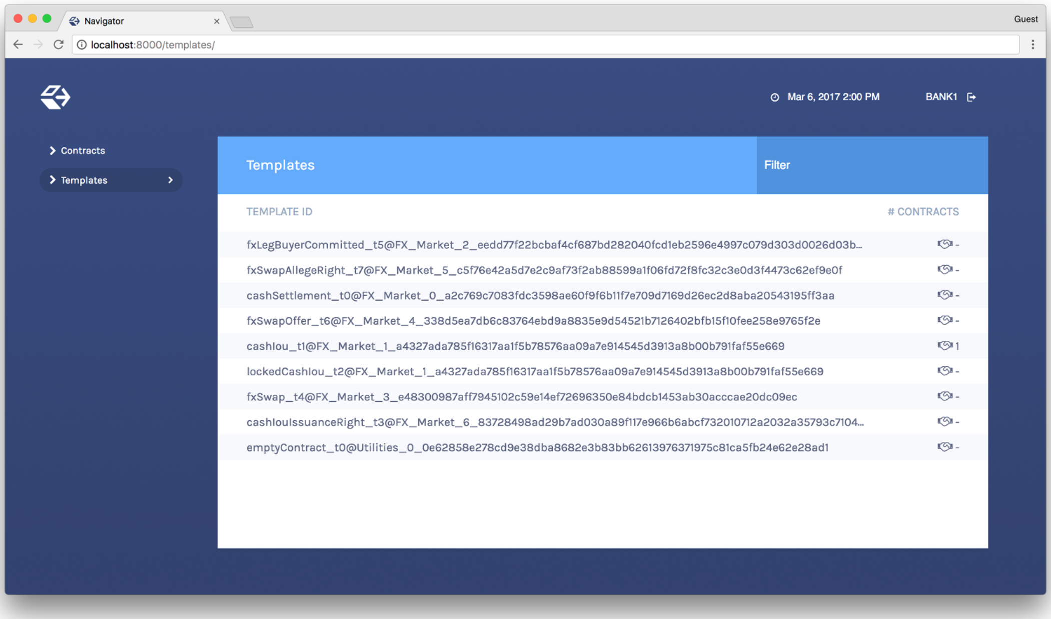
5.4.5 Viewing templates or contracts

DAML *contract templates* are models that contain the agreement statement, all the applicable parameters, and the choices that can be made in acting on that data. They specify acceptable input and the resulting output. A contract template contains placeholders rather than actual names, amounts, dates, and so on. In a *contract 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.

5.4.5.1 Listing templates

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

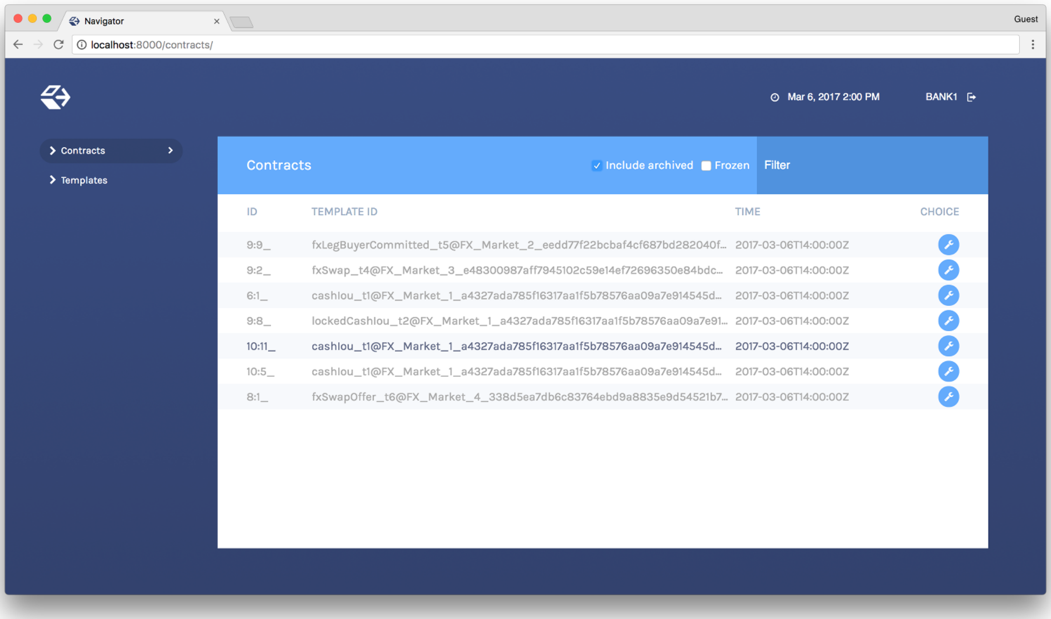


TEMPLATE ID	# CONTRACTS
fxLegBuyerCommitted_t5@FX_Market_2_eedd77f22bcbaf4cf687bd282040fcd1eb2596e4997c079d303d0026d03b…	-
fxSwapAllegeRight_t7@FX_Market_5_c5f76e42a5d7e2c9af73f2ab88599a1f06fd72f8fc32c3e0d3f4473c62ef9e0f	-
cashSettlement_t0@FX_Market_0_a2c769c7083fdc3598ae60f9f6b11f7e709d7169d26ec2d8aba20543195ff3aa	-
fxSwapOffer_t6@FX_Market_4_338d5ea7db6c83764ebd9a8835e9d54521b7126402bfb15f10fee258e9765f2e	-
cashIou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d3913a8b00b791faf55e669	1
lockedCashIou_t2@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d3913a8b00b791faf55e669	-
fxSwap_t4@FX_Market_3_e48300987aff7945102c59e14ef72696350e84bdcb1453ab30acccae20dc09ec	-
cashIouIssuanceRight_t3@FX_Market_6_83728498ad29b7ad030a89f117e966b6abcf732010712a2032a35793c7104…	-
emptyContract_t0@Utilities_0_0e62858e278cd9e38dba8682e3b83bb62613976371975c81ca5fb24e62e28ad1	-

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

5.4.5.2 Listing contracts

To view a list of available contracts, choose **Contracts** in the left pane.



ID	TEMPLATE ID	TIME	CHOICE
9:9_	fxLegBuyerCommitted_t5@FX_Market_2_eedd77f22bcbaf4cf687bd282040f...	2017-03-06T14:00:00Z	
9:2_	fxSwap_t4@FX_Market_3_e48300987aff7945102c59e14ef72696350e84bdc...	2017-03-06T14:00:00Z	
6:1_	cashIou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d...	2017-03-06T14:00:00Z	
9:8_	lockedCashIou_t2@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e91...	2017-03-06T14:00:00Z	
10:11_	cashIou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d...	2017-03-06T14:00:00Z	
10:5_	cashIou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d...	2017-03-06T14:00:00Z	
8:1_	fxSwapOffer_t6@FX_Market_4_338d5ea7db6c83764ebd9a8835e9d54521b7...	2017-03-06T14:00:00Z	

In the Contracts list:

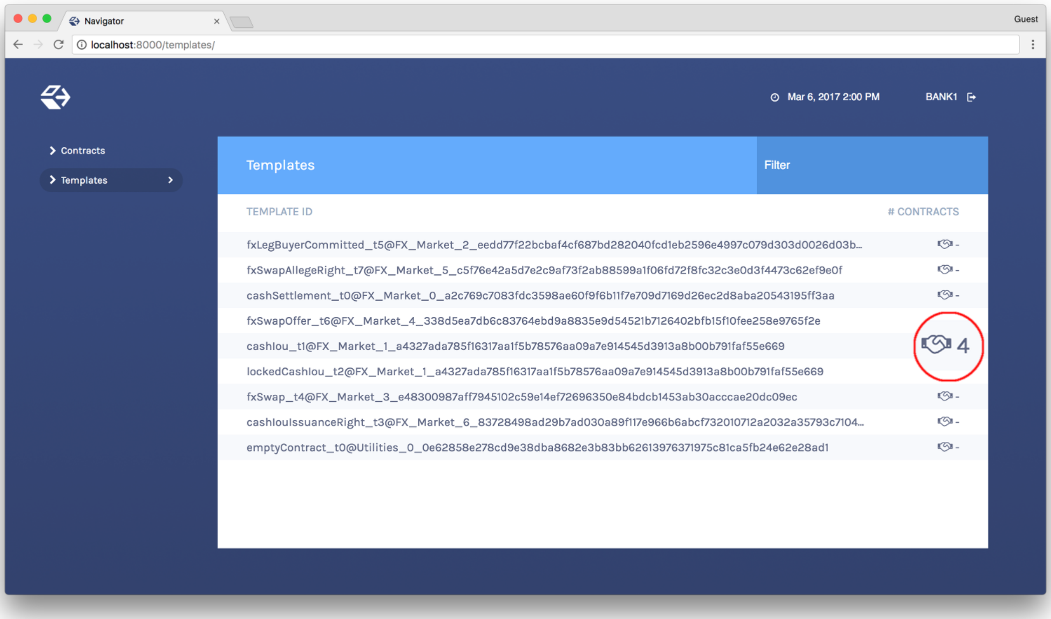
> Changes to the ledger are automatically reflected in the list of contracts. To avoid the automatic updates, select the **Frozen** checkbox. Contracts will still be marked as archived, but the contracts list will not change.
>
> Filter the displayed contracts by entering text in the **Filter** field at the top right.
> Use the **Include Archived** checkbox at the top to include or exclude archived contracts.

5.4.5.3 Viewing contracts based on a template

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

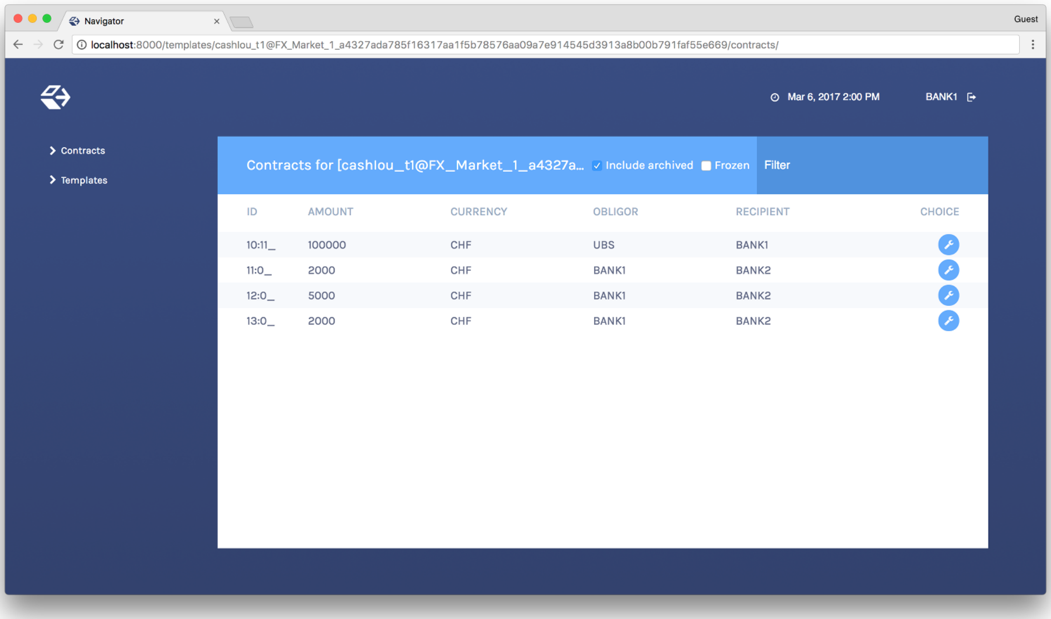
1. You will see icons to the right of template IDs in the template list with a number indicating how many contracts are based on this template.
2. Click the number to display a list of contracts based on that template.

Number of Contracts



TEMPLATE ID	# CONTRACTS
fxLegBuyerCommitted_t5@FX_Market_2_eedd77f22bcbaf4cf687bd282040fcd1eb2596e4997c079d303d0026d03b...	-
fxSwapAllegeRight_t7@FX_Market_5_c5f76e42a5d7e2c9af73f2ab88599a1f06fd72f8fc32c3e0d3f4473c62ef9e0f	-
cashSettlement_t0@FX_Market_0_a2c769c7083fdc3598ae60f9f6b11f7e709d7169d26ec2d8aba20543195ff3aa	-
fxSwapOffer_t6@FX_Market_4_338d5ea7db6c83764ebd9a8835e9d54521b7126402bfb15f10fee258e9765f2e	
cashIou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d3913a8b00b791faf55e669	4
lockedCashIou_t2@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d3913a8b00b791faf55e669	
fxSwap_t4@FX_Market_3_e48300987aff7945102c59e14ef72696350e84bdcb1453ab30acccae20dc09ec	-
cashIouIssuanceRight_t3@FX_Market_6_83728498ad29b7ad030a89f117e966b6abcf732010712a2032a35793c7104...	-
emptyContract_t0@Utilities_0_0e62858e278cd9e38dba8682e3b83bb62613976371975c81ca5fb24e62e28ad1	-

List of Contracts

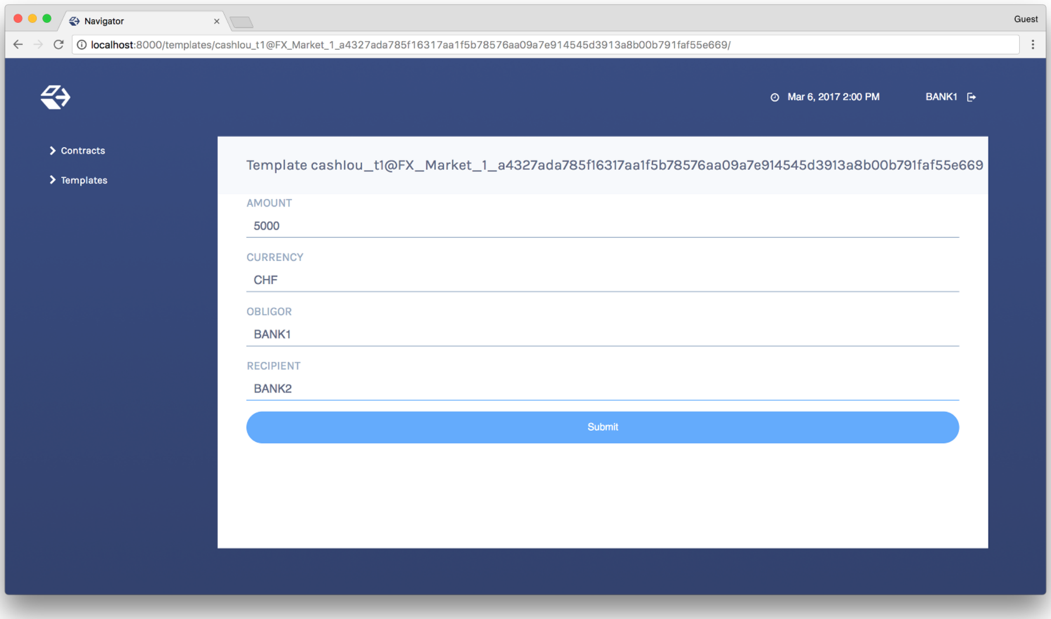

Navigator
Guest
localhost:8000/templates/cashIou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d3913a8b00b791faf55e669/contracts/
Mar 6, 2017 2:00 PM
BANK1
Contracts
Templates
Contracts for [cashIou_t1@FX_Market_1_a4327a...
☑ Include archived
□ Frozen
Filter

ID	AMOUNT	CURRENCY	OBLIGOR	RECIPIENT	CHOICE
10:11_	100000	CHF	UBS	BANK1	
11:0_	2000	CHF	BANK1	BANK2	
12:0_	5000	CHF	BANK1	BANK2	
13:0_	2000	CHF	BANK1	BANK2	

5.4.5.4 Viewing template and contract details

To view template or contract details, click on a template or contract in the list. The template or contracts detail page is displayed.

Template Details



localhost:8000/templates/cashlou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d3913a8b00b791faf55e669/

Template cashlou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d3913a8b00b791faf55e669

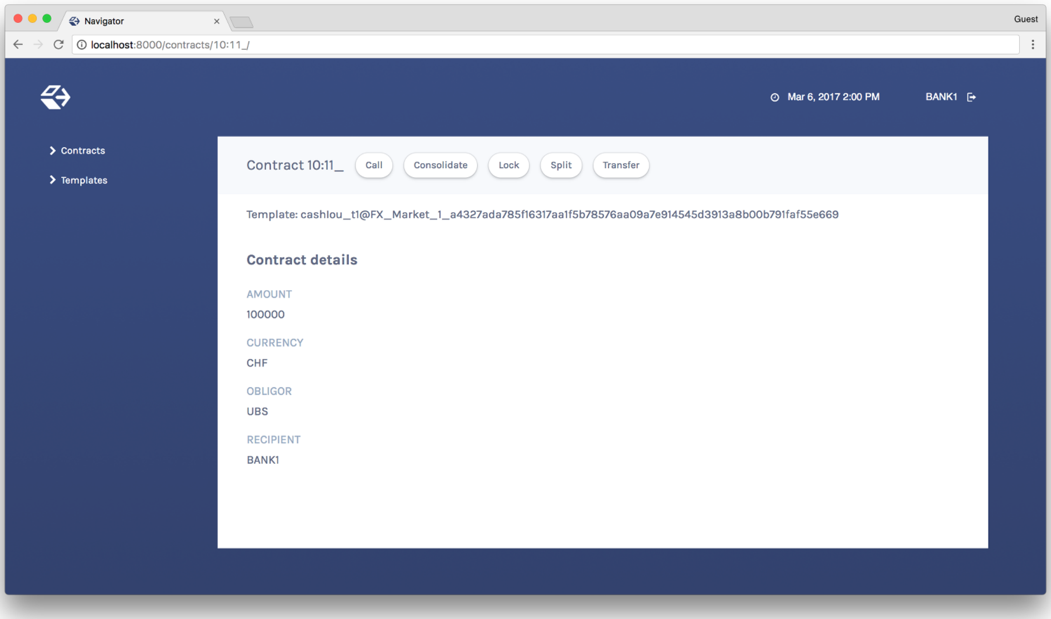
AMOUNT
5000

CURRENCY
CHF

OBLIGOR
BANK1

RECIPIENT
BANK2

Contract Details



localhost:8000/contracts/10:11_/

Contract 10:11_

Template: cashlou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d3913a8b00b791faf55e669

Contract details

AMOUNT
100000

CURRENCY
CHF

OBLIGOR
UBS

RECIPIENT
BANK1

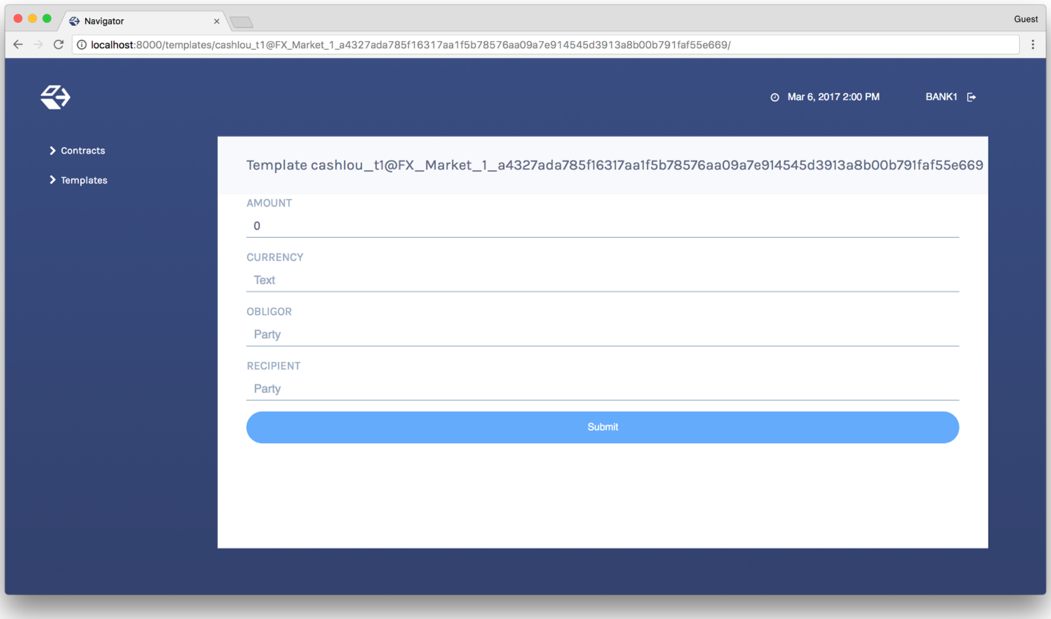
5.4.6 Using Navigator

5.4.6.1 Creating contracts

Contracts in a ledger are created automatically when you exercise choices. In some cases, you create a contract directly from a template. This feature can be particularly useful for testing and 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.


Navigator
Guest
localhost:8000/templates/cashIou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d3913a8b00b791faf55e669/
Mar 6, 2017 2:00 PM
BANK1
Contracts
Templates
Submit

Template cashIou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d3913a8b00b791faf55e669

AMOUNT
0

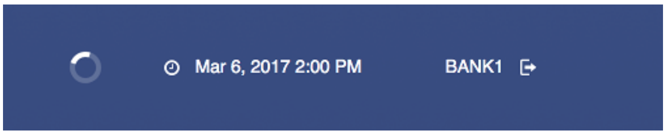
CURRENCY
Text

OBLIGOR
Party

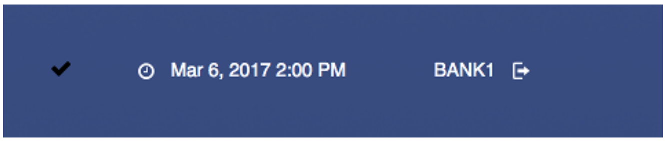
RECIPIENT
Party

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



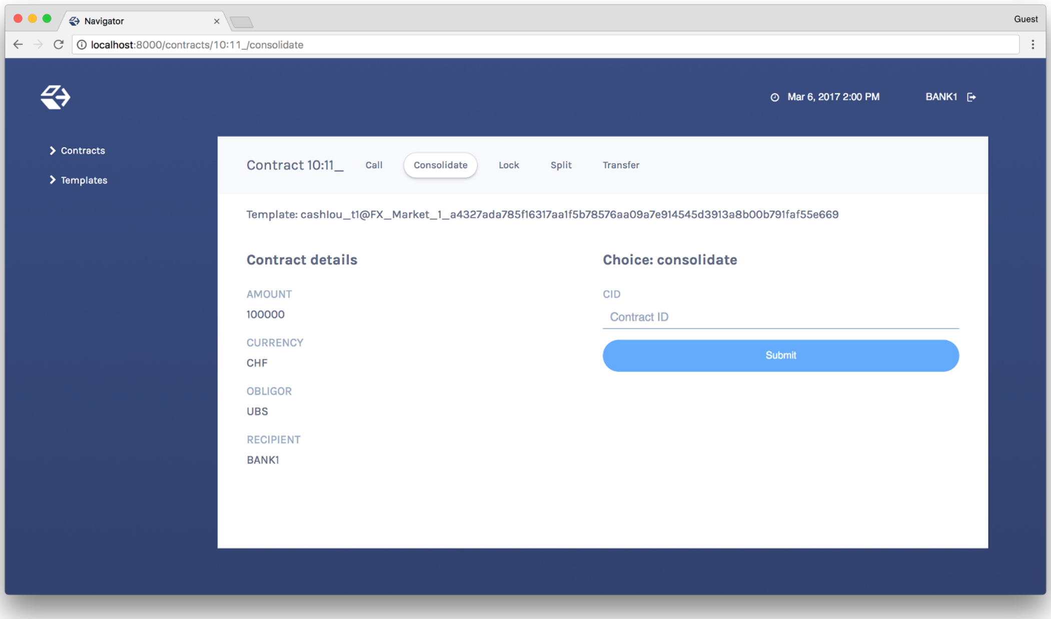
When committed to the ledger



5.4.6.2 Exercising choices

To exercise a choice:

1. Navigate to the contract details page (see above).
2. Click the choice you want to exercise in the choice list.
3. Complete the form.
4. Choose the **Submit** button.



Contract 10:11_

Template: cashlou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d3913a8b00b791faf55e669

Contract details

AMOUNT
100000

CURRENCY
CHF

OBLIGOR
UBS

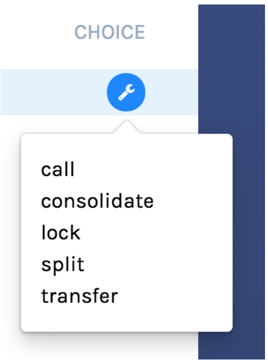
RECIPIENT
BANK1

Choice: consolidate

CID
Contract ID

Or

1. Navigate to the choice form by clicking the wrench icon in a contract list.
2. Select a choice.



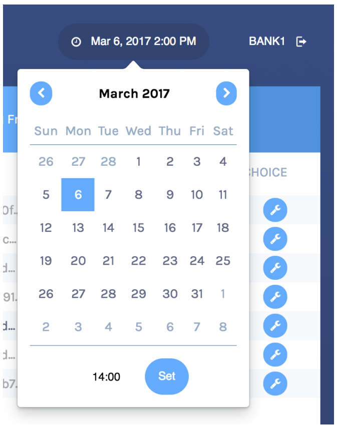
You will see the loading and confirmation indicators, as pictured above in Creating Contracts.

5.4.6.3 Advancing time

It is possible to advance time against the DAML Sandbox. (This is not true of all DAML Ledgers.) This advance-time functionality can be useful when testing, for example, when entering a trade on one date and settling it on a later date.

To advance time:

1. Click on the ledger time indicator in the navbar at the top of the screen.
2. Select a new date / time.
3. Choose the **Set** button.



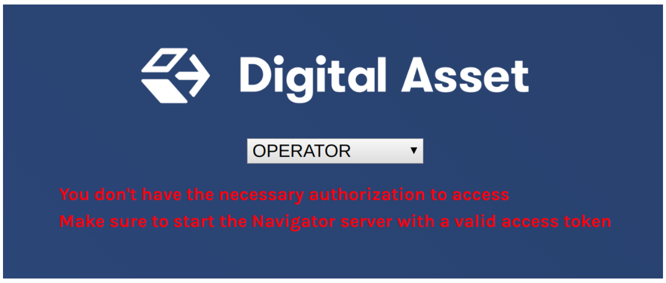
5.4.7 Authorizing Navigator

If you are running Navigator against a Ledger API server that verifies authorization, you must provide the access token when you start the Navigator server.

The access token retrieval depends on the specific DAML setup you are working with: please refer to the ledger operator to learn how.

Once you have retrieved your access token, you can provide it to Navigator by storing it in a file and provide the path to it using the `--access-token-file` command line option.

If the access token cannot be retrieved, is missing or wrong, you'll be unable to move past the Navigator's frontend login screen and see the following:



You don't have the necessary authorization to access
Make sure to start the Navigator server with a valid access token

5.4.8 Advanced usage

5.4.8.1 Customizable table views

Customizable table views is an advanced rapid-prototyping feature, intended for DAML developers who wish to customize the Navigator UI without developing a custom application.

To use customized table views:

1. Create a file `frontend-config.js` in your project root folder (or the folder from which you run Navigator) with the content below:

```
import { DamlLfValue } from '@da/ui-core';

export const version = {
  schema: 'navigator-config',
  major: 2,
  minor: 0,
};
```

(continues on next page)

(continued from previous page)

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

2. Reload your Navigator browser tab. You should now see a sidebar item titled Filtered contracts that links to a table with contracts filtered and sorted by ID.

To debug config file errors and learn more about the config file API, open the Navigator `/config` page in your browser (e.g., http://localhost:7500/config).

5.4.8.2 Using Navigator with a DAML Ledger

By default, Navigator is configured to use an unencrypted connection to the ledger. To run Navigator against a secured DAML Ledger, configure TLS certificates using the `--pem`, `--crt`, and `--cacrt` command line parameters. Details of these parameters are explained in the command line help:

```
daml navigator --help
```

Chapter 6

Background concepts

6.1 Glossary of concepts

6.1.1 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.

6.1.1.1 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.

6.1.1.2 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*.

6.1.1.3 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.

6.1.1.4 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*.

6.1.1.5 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.

6.1.1.6 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/stdlib/index.

6.1.1.7 Agreement

An **agreement** is part of a *contract*. It is text that explains what the contract represents.

It can be used to clarify the legal intent of a contract, but this text isn't evaluated programmatically.

See *Reference: templates*.

6.1.1.8 Create

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

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

A *party submits* a create *command*.

See *Reference: updates*.

6.1.1.9 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*.

6.1.1.10 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*
- ensuring that parties can exclusively create contracts, observe contracts, and exercise choices that they are meant to
- acting as regression tests to confirm that everything keeps working correctly

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*.

6.1.1.11 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*.

6.1.1.12 DAR file, DALF file

A `.dar` file is the result of compiling DAML using the *Assistant*.

You upload `.dar` files to a *ledger* in order to be able to create contracts from the templates in that file.

A `.dar` contains multiple `.dalf` files. A `.dalf` file is the output of a compiled DAML package or library. Its underlying format is *DAML-LF*.

6.1.2 SDK tools

6.1.2.1 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)*.

6.1.2.2 Studio

DAML Studio is a plugin for Visual Studio Code, and is the IDE for writing DAML code.

See *DAML Studio*.

6.1.2.3 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*.

6.1.2.4 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*.

6.1.2.5 Extractor

Extractor is a tool for extracting contract data for a single party into a PostgreSQL database.

See *Extractor*.

6.1.3 Building applications

6.1.3.1 Application, ledger client, integration

Application, **ledger client** and **integration** are all terms for an application that sits on top of the *ledger*. These usually *read from the ledger*, *send commands* to the ledger, or both.

There's a lot of information available about application development, starting with the *Application architecture* page.

6.1.3.2 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*.

6.1.3.3 Ledger API libraries

The following libraries wrap the *ledger API* for more native experience applications development.

Java bindings

An idiomatic Java library for writing *ledger applications*. See *Java bindings*.

Scala bindings

An idiomatic Scala library for writing *ledger applications*. See *Scala bindings*.

gRPC API

The low-level ledger API that all of the other bindings use. Written in gRPC. See *gRPC*.

6.1.3.4 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.

6.1.3.5 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*.

6.1.3.6 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)

6.1.4 General concepts

6.1.4.1 Ledger, DAML ledger

Ledger can refer to a lot of things, but a ledger is essentially the underlying storage mechanism for a running DAML applications: it's where the contracts live. A **DAML ledger** is a ledger that you can store DAML contracts on, because it implements the *ledger API*.

DAML ledgers provide various guarantees about what you can expect from it, all laid out in the *DAML Ledger Model* page.

When you're developing, you'll use *Sandbox* as your ledger.

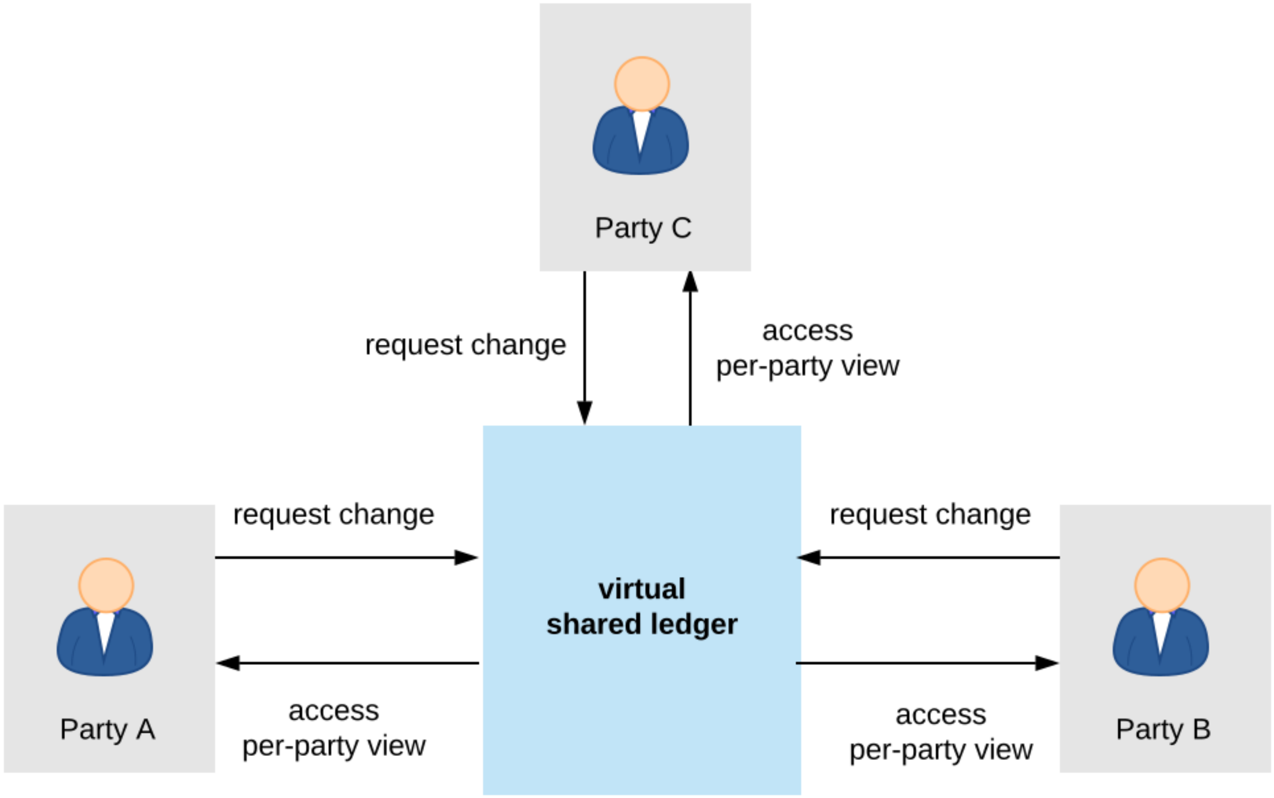
If you would like to integrate DAML with a storage infrastructure not already in development (see daml.com), please get in touch on Slack in the channel `#daml-contributors`.

6.1.4.2 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.

6.2 DAML Ledger Model

DAML Ledgers enable multi-party workflows by providing parties with a virtual *shared ledger*, which encodes the current state of their shared contracts, written in DAML. At a high level, the interactions are visualized as follows:



The DAML ledger model defines:

1. what the ledger looks like - the structure of DAML ledgers
2. who can request which changes - the integrity model for DAML ledgers
3. who sees which changes and data - the privacy model for DAML ledgers

The below sections review these concepts of the ledger model in turn. They also briefly describe the link between DAML and the model.

6.2.1 Structure

This section looks at the structure of a DAML ledger and the associated ledger changes. The basic building blocks of changes are *actions*, which get grouped into *transactions*.

6.2.1.1 Actions and Transactions

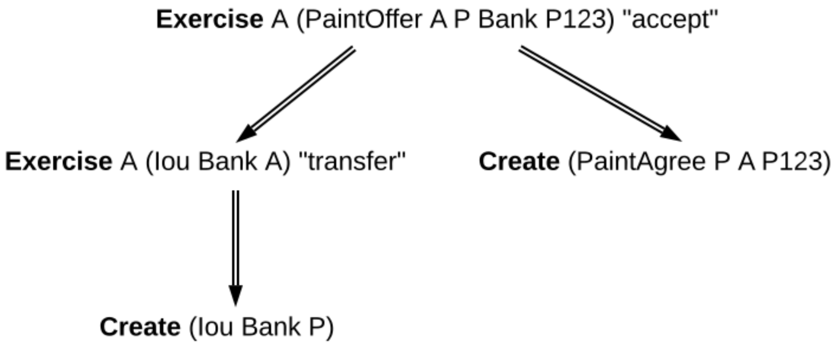
One of the main features of the DAML ledger model is a *hierarchical action structure*.

This structure is illustrated below on a toy example of a multi-party interaction. Alice (*A*) gets some digital cash, in the form of an I-Owe-You (IOU for short) from a bank, and she needs her house painted. She gets an offer from a painter (*P*) with reference number *P123* to paint her house in exchange for this IOU. Lastly, *A* accepts the offer, 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 DAML ledger model.

The visual notation below records the relations between the actions during the above acceptance.


Exercise A (PaintOffer A P Bank P123) "accept"
Exercise A (Iou Bank A) "transfer"
Create (PaintAgree P A P123)
Create (Iou Bank P)

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.

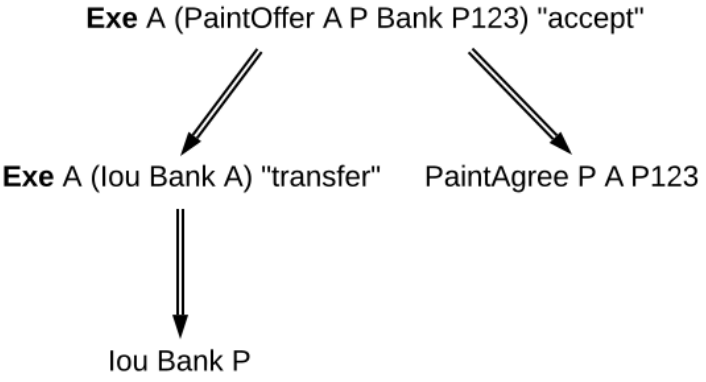
```
Action       ::= 'Create' contract
               | 'Exercise' party* contract Kind Transaction
               | 'Fetch' party* contract
               | 'NoSuchKey' key
Transaction  ::= Action*
Kind         ::= 'Consuming' | 'NonConsuming'
```

The visual notation presented earlier captures actions precisely with conventions that:

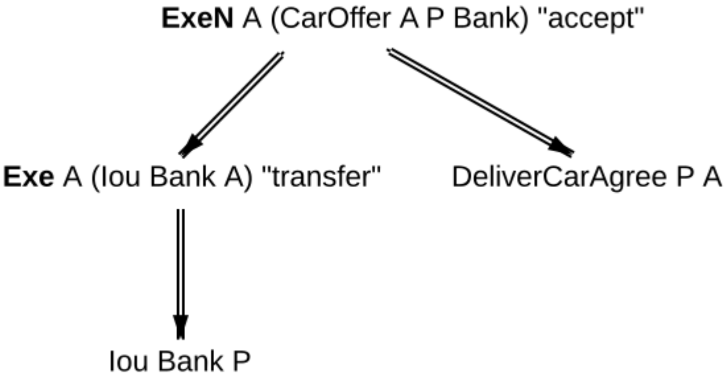
1. **Exercise** denotes consuming, **ExerciseN** non-consuming exercises, and **Fetch** a fetch.

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.

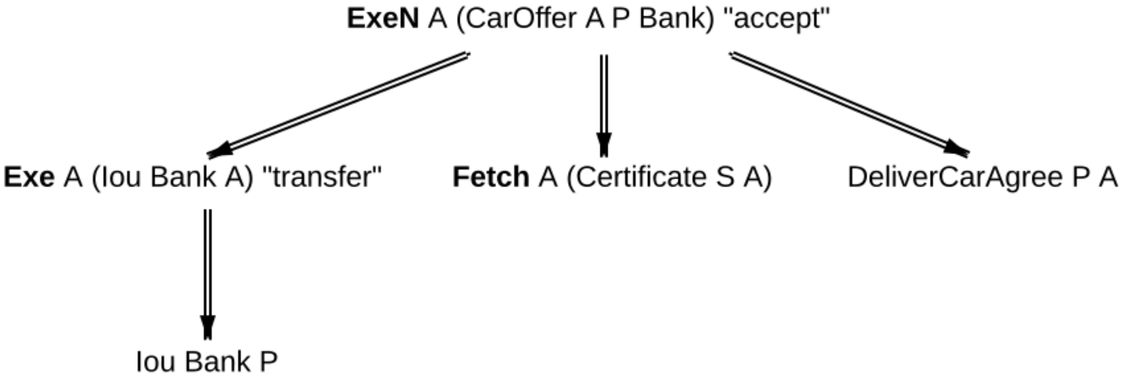
An alternative shorthand notation, shown below uses the abbreviations **Exe** and **ExeN** for exercises, and omits the **Create** labels on create actions.


Exe A (PaintOffer A P Bank P123) "accept"
Exe A (Iou Bank A) "transfer"
PaintAgree P A P123
Iou Bank P

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.


ExeN A (CarOffer A P Bank) "accept"
Exe A (Iou Bank A) "transfer"
DeliverCarAgree P A
Iou Bank P

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*.


ExeN A (CarOffer A P Bank) "accept"
Exe A (Iou Bank A) "transfer"
Fetch A (Certificate S A)
DeliverCarAgree P A
Iou Bank P

In the paint offer example, the underlying type of contracts consists of three sorts of contracts:

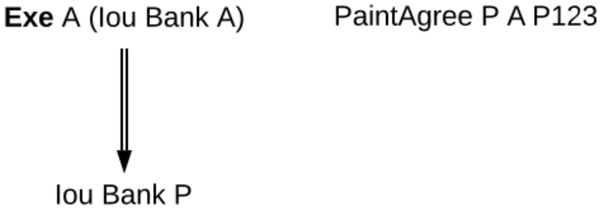
PaintOffer houseOwner painter obligor refNo Intuitively an offer (with a reference number) by which the painter proposes to the house owner to paint her house, in exchange for a single IOU token issued by the specified obligor.

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).

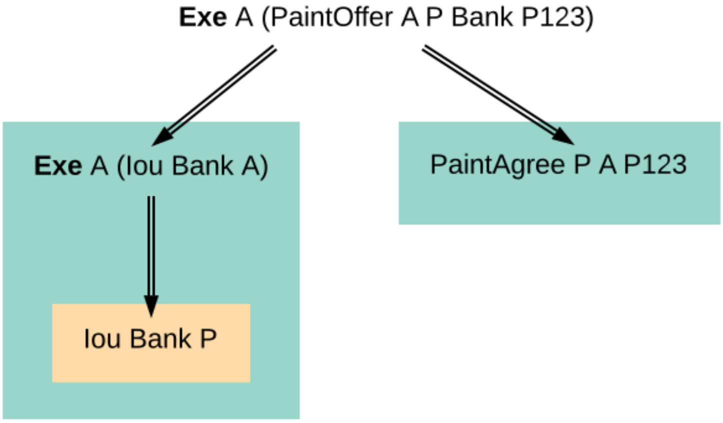
In practice, multiple IOU contracts would exist between the same *obligor* and *owner*, in which case each contract should have a unique identifier. However, in this section, each contract only appears once, allowing us to drop the notion of identifiers for simplicity reasons.

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.

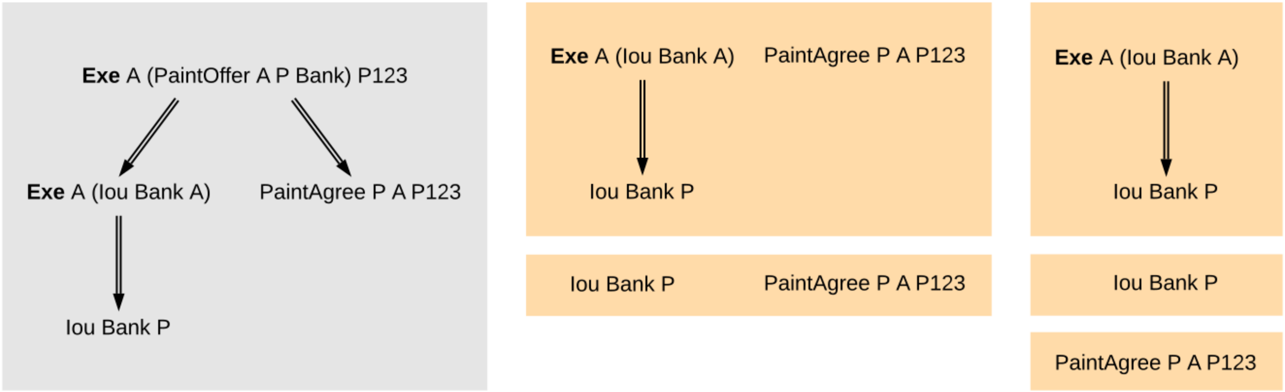


For an action *act*, its **proper subactions** are all actions in the consequences of *act*, together with all of their proper subactions. Additionally, *act* is a (non-proper) **subaction** of itself.

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 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.

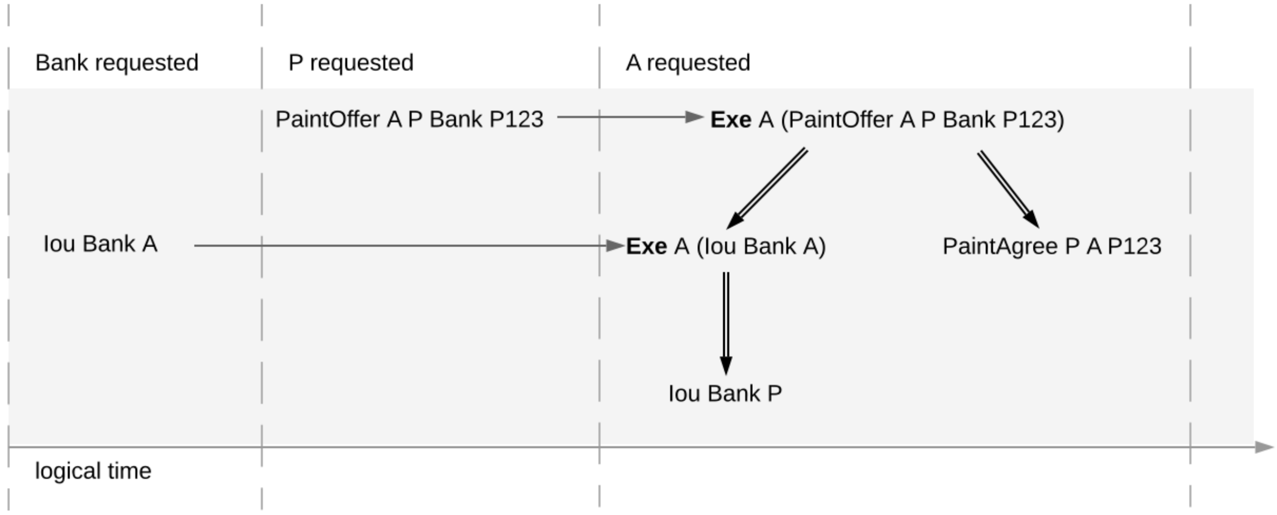
6.2.1.2 Ledgers

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 DAML Ledger API offers the request functionality only to individual parties. Given a commit *(p, tx)* with transaction $tx = act_1, \ldots, act_n$, every act_i is called a **top-level action** of the commit. A **ledger** is a sequence of commits. A top-level action of any ledger commit is also a top-level action of the ledger.

The following EBNF grammar summarizes the structure of commits and ledgers:

```
Commit   ::= party Transaction
Ledger   ::= Commit*
```

A DAML ledger thus represents the full history of all actions taken by parties.[1] Since the ledger is a sequence (of dependent actions), it induces an *order* on the commits in the ledger. Visually, a ledger can be represented as a sequence growing from left to right as time progresses. Below, dashed vertical lines mark the boundaries of commits, and each commit is annotated with its requester(s). Arrows link the create and exercise actions on the same contracts. These additional arrows highlight that the ledger forms a **transaction graph**. For example, the aforementioned house painting scenario is visually represented as follows.



The definitions presented here are all the ingredients required to *record* the interaction between parties in a DAML ledger. That is, they address the first question: what do changes and ledgers look like? . To answer the next question, who can request which changes , a precise definition is needed of which ledgers are permissible, and which are not. For example, the above paint offer ledger is intuitively permissible, while all of the following ledgers are not.

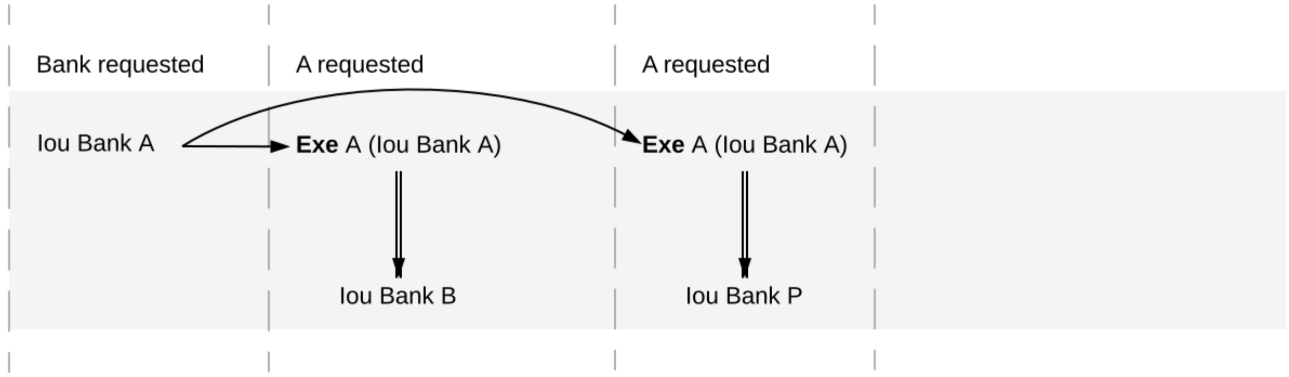


Fig. 1: Alice spending her IOU twice (double spend), once transferring it to *B* and once to *P*.

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.

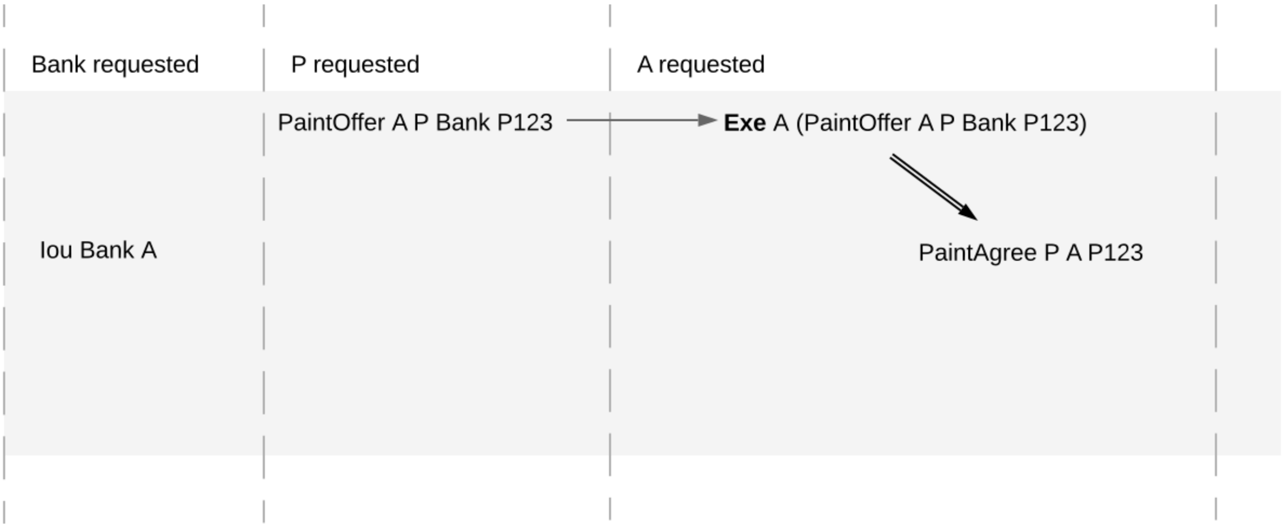


Fig. 2: Alice changing the offer's outcome by removing the transfer of the *Iou*.

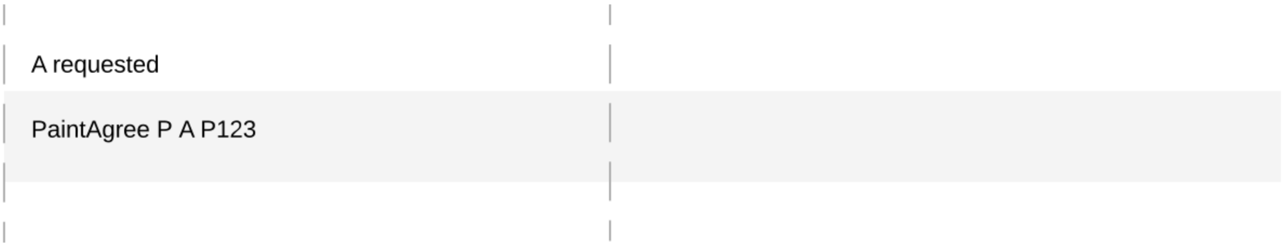


Fig. 3: An obligation imposed on the painter without his consent.

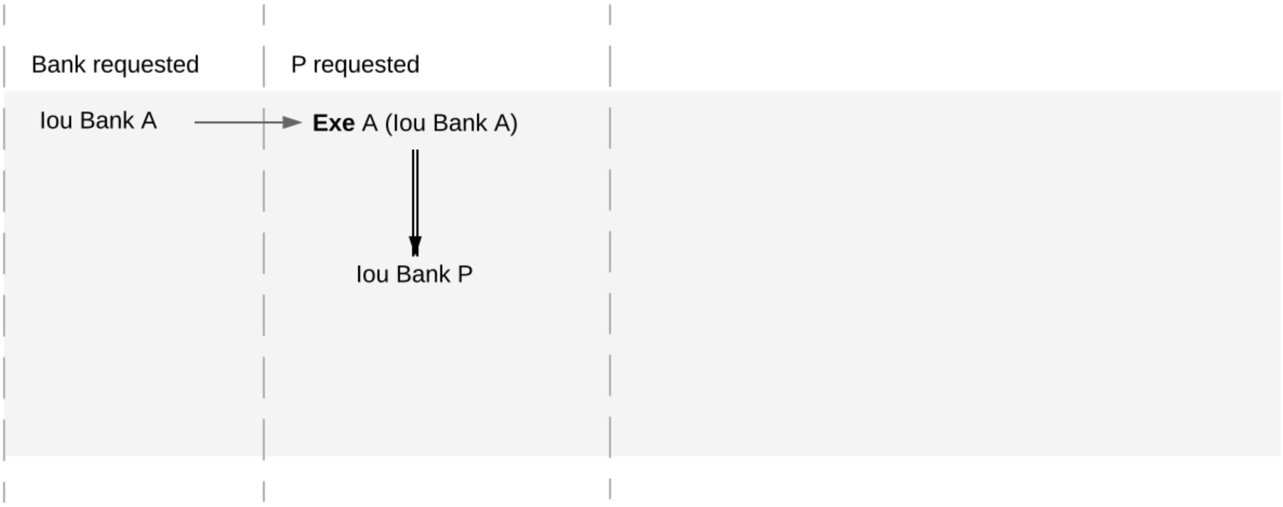


Fig. 4: Painter stealing Alice's IOU. Note that the ledger would be intuitively permissible if it was Alice performing the last commit.

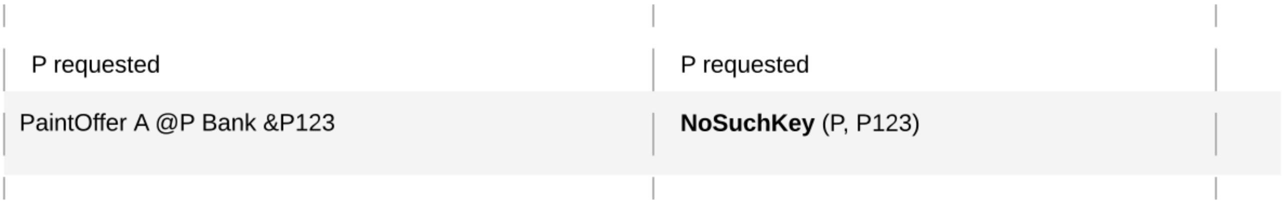


Fig. 5: Painter falsely claiming that there is no offer.

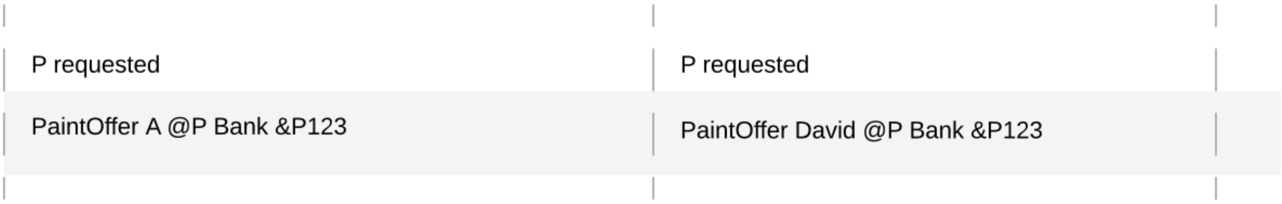


Fig. 6: Painter trying to create two different paint offers with the same reference number.

6.2.2 Integrity

This section addresses the question of who can request which changes.

6.2.2.1 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.

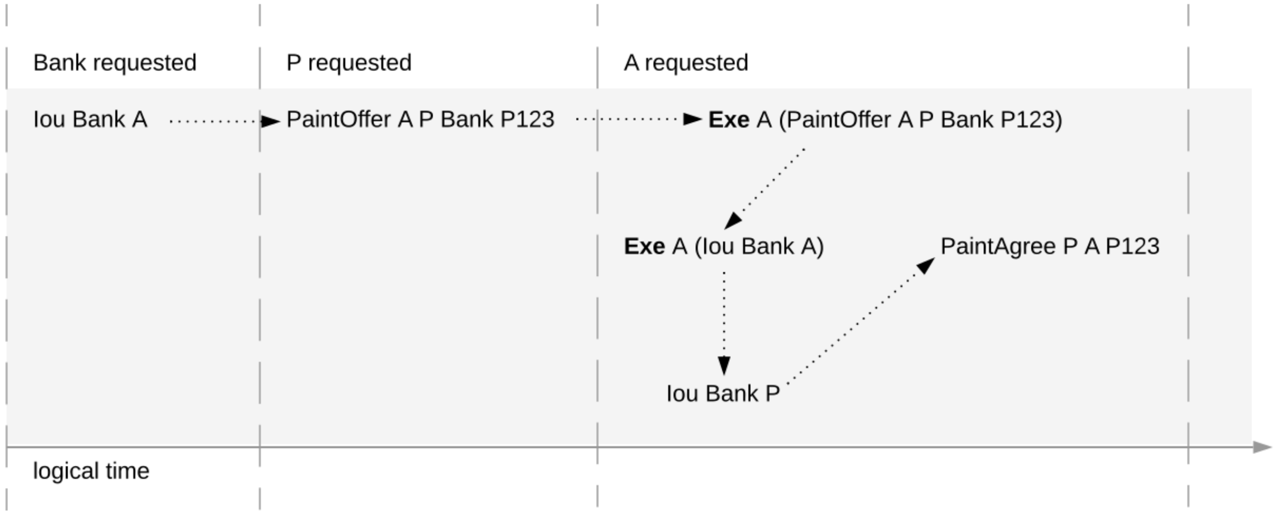
Only the last of these conditions depends on the party (or parties) requesting the change; the other two are general.

6.2.2.2 Consistency

Consistency consists of two parts:

1. *Contract consistency*: Contracts must be created before they are used, and they cannot be used once they are consumed.
2. *Key consistency*: Keys are unique and key assertions are satisfied.

To define this precisely, notions of before and after are needed. These are given by putting all actions in a sequence. Technically, the sequence is obtained by a pre-order traversal of the ledger's actions, noting that these actions form an (ordered) forest. Intuitively, it is obtained by always picking parent actions before their proper subactions, and otherwise always picking the actions on the left before the actions on the right. The image below depicts the resulting order on the paint offer example:



In the image, an action *act* happens before action *act'* if there is a (non-empty) path from *act* to *act'*. Then, *act'* happens after *act*.

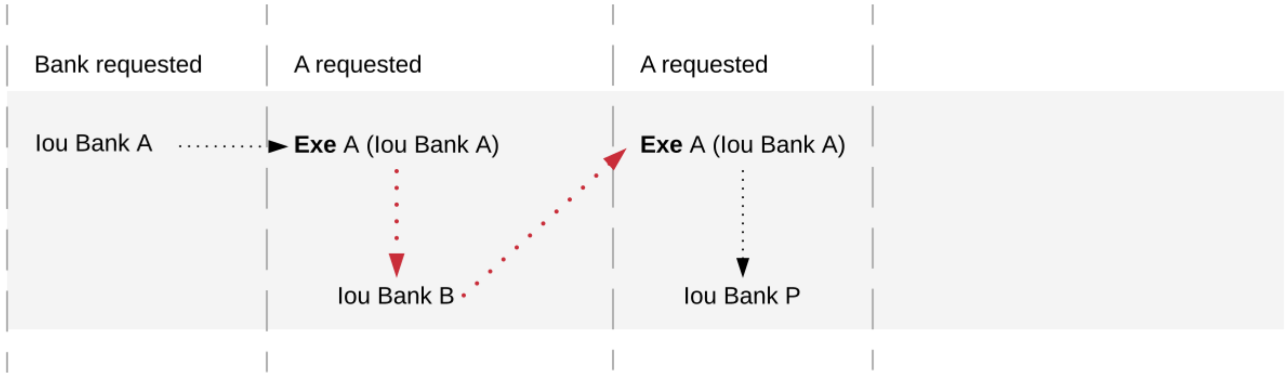
Contract consistency

Contract consistency ensures that contracts are used after they have been created and before they are consumed.

Definition contract consistency A ledger is **consistent for a contract c** if all of the following holds for all actions *act* on *c*:

1. either *act* is itself **Create c** or a **Create c** happens before *act*
2. *act* does not happen before any **Create c** action
3. *act* does not happen after any **Exercise** action consuming *c*.

The consistency condition rules out the double spend example. As the red path below indicates, the second exercise in the example happens after a consuming exercise on the same contract, violating the contract consistency criteria.



In addition to the consistency notions, the before-after relation on actions can also be used to define the notion of **contract state** at any point in a given transaction. The contract state is changed by creating the contract and by exercising it consumingly. At any point in a transaction, we can then define the latest state change in the obvious way. Then, given a point in a transaction, the contract state of *c* is:

1. **active**, if the latest state change of *c* was a create;
2. **archived**, if the latest state change of *c* was a consuming exercise;

3. **inexistent**, if *c* never changed state.

A ledger is consistent for *c* exactly if **Exercise** and **Fetch** actions on *c* happen only when *c* is active, and **Create** actions only when *c* is inexistent. The figures below visualize the state of different contracts at all points in the example ledger.

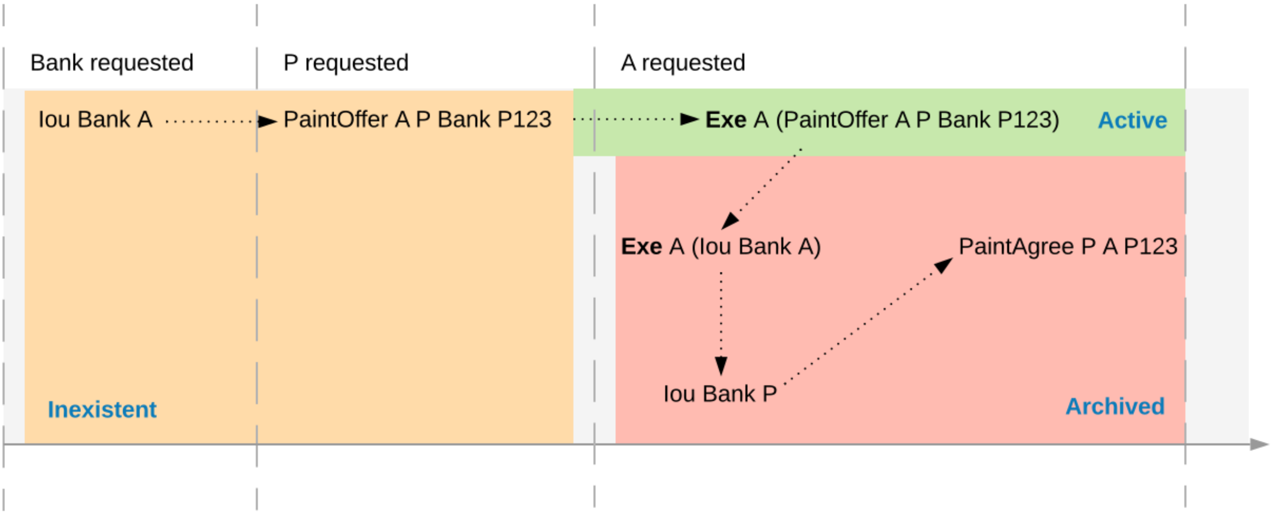


Fig. 7: Activeness of the *PaintOffer* contract

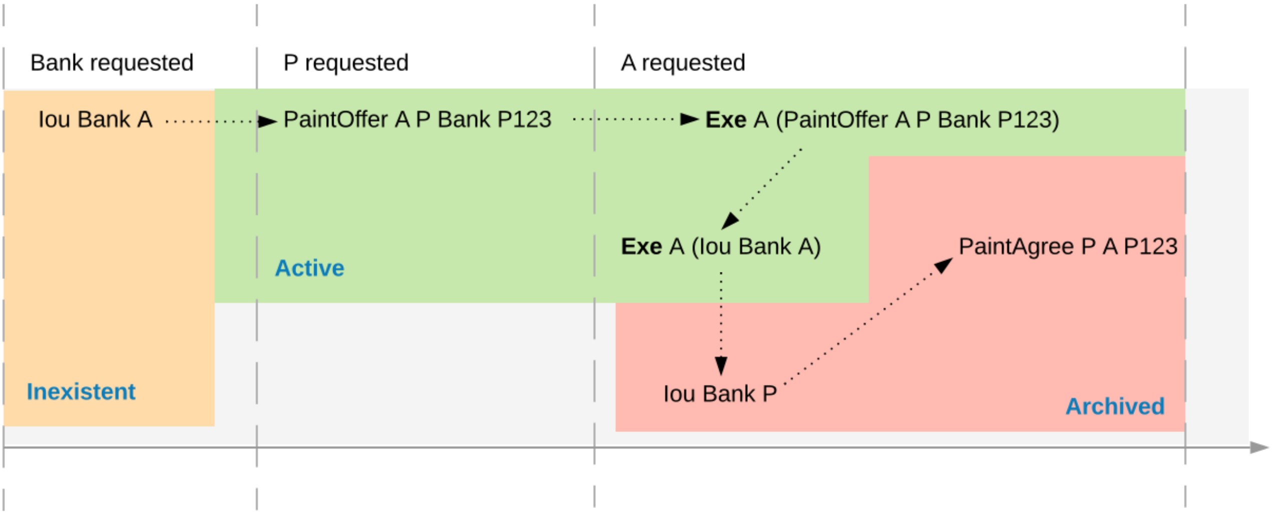


Fig. 8: Activeness of the *Iou Bank A* contract

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**.

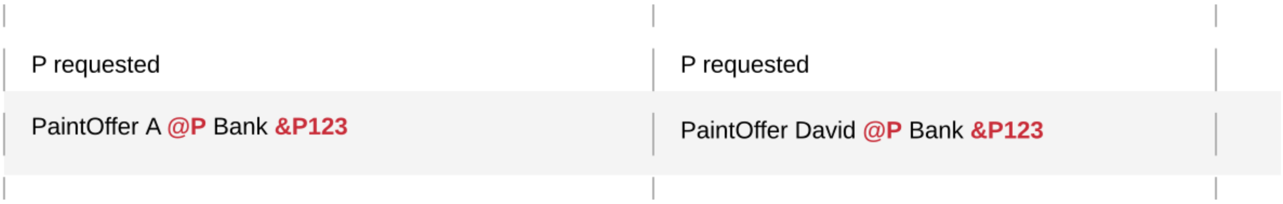
A key is **unassigned** if its key state is either **free** or **unknown**.

Key consistency ensures that there is at most one active contract for each key and that all key assertions are satisfied.

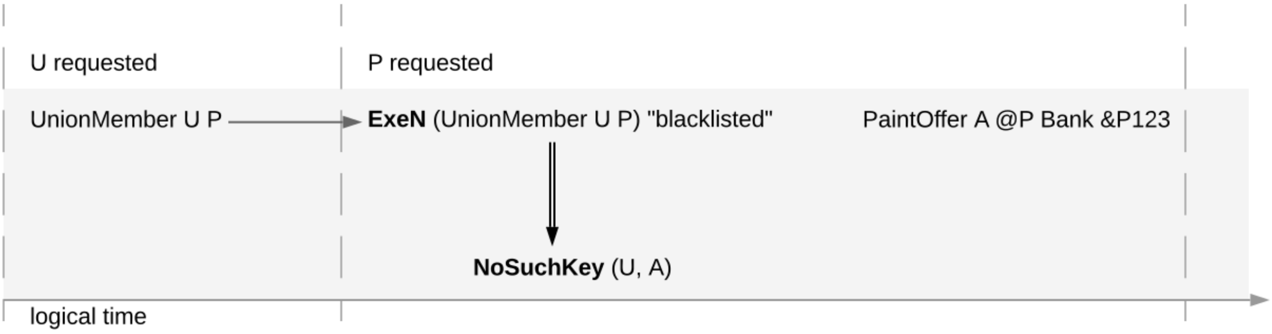
Definition key consistency A ledger is **consistent for a key** *k* if for every action *act* on *k*, the key state *s* before *act* satisfies

> If *act* is a **Create** action or **NoSuchKey** assertion, then *s* is **free** or **unknown**.
> If *act* is an **Exercise** or **Fetch** action on some contract *c*, then *s* is **assigned** to *c* or **unknown**.

Key consistency rules out the problematic examples around key consistency. For example, suppose that the painter *P* has made a paint offer to *A* with reference number *P123*, but *A* has not yet accepted it. When *P* tries to create another paint offer to *David* with the same reference number *P123*, then this creation action would violate key uniqueness. The following ledger violates key uniqueness for the key *(P, P123)*.



Key assertions can be used in workflows to evidence the inexistence of a certain kind of contract. For example, suppose that the painter *P* is a member of the union of painters *U*. This union maintains a blacklist of potential customers that its members must not do business with. A customer *A* is considered to be on the blacklist if there is an active contract *Blacklist @U &A*. To make sure that the painter *P* does not make a paint offer if *A* is blacklisted, the painter combines its commit with a **NoSuchKey** assertion on the key *(U, A)*. The following ledger shows the transaction, where *UnionMember U P* represents *P*'s membership in the union *U*. It grants *P* the choice to perform such an assertion, which is needed for *authorization*.



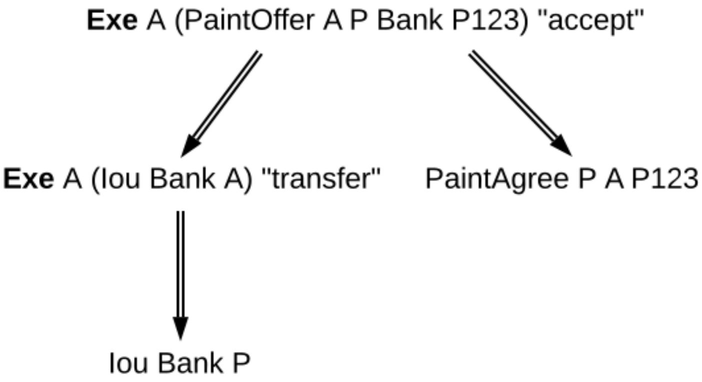
Key consistency extends to actions, transactions and lists of transactions just like the other consistency notions.

Ledger consistency

Definition ledger consistency A ledger is **consistent** if it is consistent for all contracts and for all keys.

Internal consistency

The above consistency requirement is too strong for actions and transactions in isolation. For example, the acceptance transaction from the paint offer example is not consistent as a ledger, because *PaintOffer A P Bank* and the *Iou Bank A* contracts are used without being created before:


Exe A (PaintOffer A P Bank P123) "accept"
Exe A (Iou Bank A) "transfer"
PaintAgree P A P123
Iou Bank P

However, the transaction can still be appended to a ledger that creates these contracts and yields a consistent ledger. Such transactions are said to be internally consistent, and contracts such as the *PaintOffer A P Bank P123* and *Iou Bank A* are called input contracts of the transaction. Dually, output contracts of a transaction are the contracts that a transaction creates and does not archive.

Definition internal consistency for a contract A transaction is **internally consistent for a contract c** if the following holds for all of its subactions *act* on the contract *c*

1. *act* does not happen before any **Create c** action
2. *act* does not happen after any exercise consuming *c*.

A transaction is **internally consistent** if it is internally consistent for all contracts and consistent for all keys.

Definition input contract For an internally consistent transaction, a contract *c* is an **input contract** of the transaction if the transaction contains an **Exercise** or a **Fetch** action on *c* but not a **Create c** action.

Definition output contract For an internally consistent transaction, a contract *c* is an **output contract** of the transaction if the transaction contains a **Create c** action, but not a consuming **Exercise** action on *c*.

Note that the input and output contracts are undefined for transactions that are not internally consistent. The image below shows some examples of internally consistent and inconsistent transactions.

Similar to input contracts, we define the input keys as the set that must be unassigned at the beginning of a transaction.

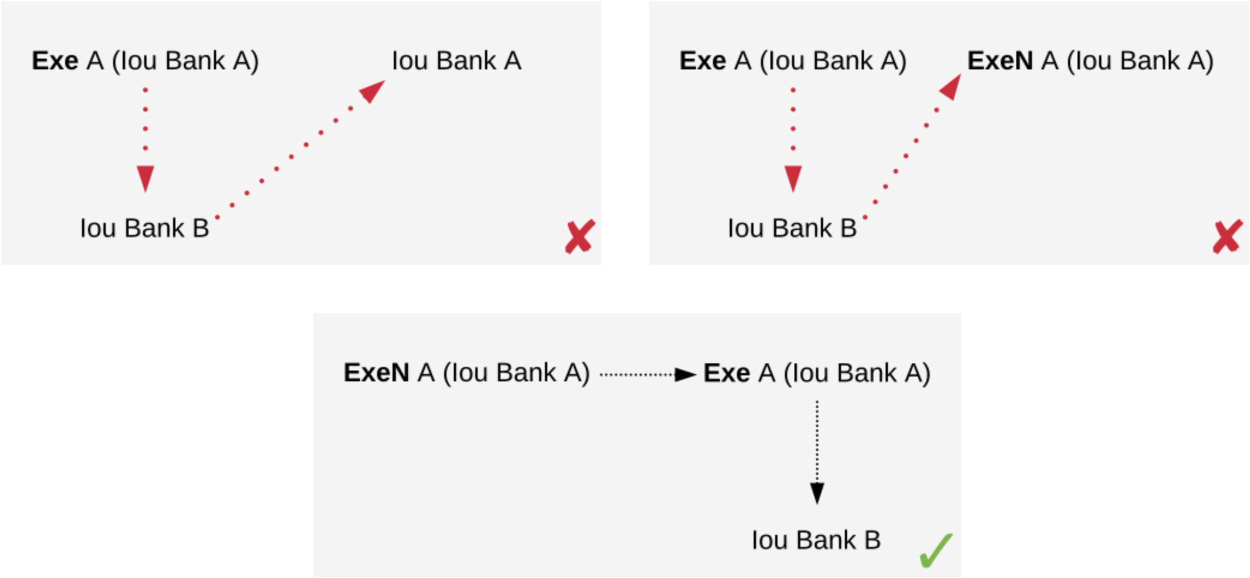


Fig. 9: The first two transactions violate the conditions of internal consistency. The first transaction creates the *Iou* after exercising it consumingly, violating both conditions. The second transaction contains a (non-consuming) exercise on the *Iou* after a consuming one, violating the second condition. The last transaction is internally consistent.

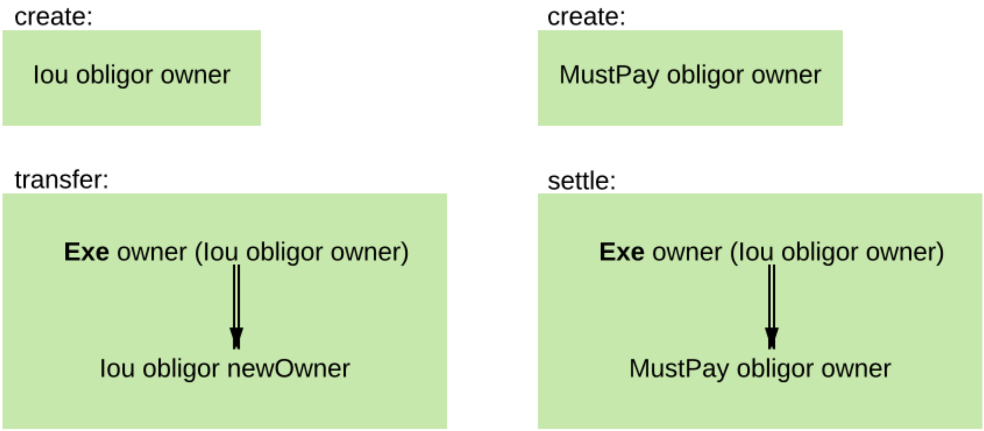
Definition input key A key *k* is an **input key** to an internally consistent transaction if the first action *act* on *k* is either a **Create** action or a **NoSuchKey** assertion.

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.

6.2.2.3 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.

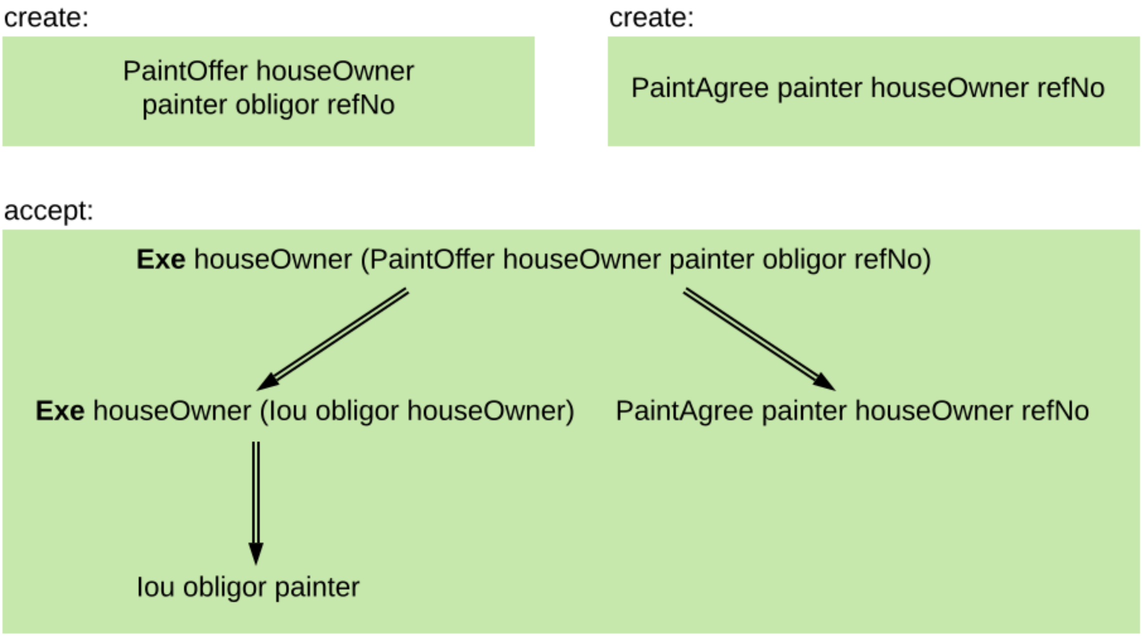
For example, the set of allowed actions on IOU contracts could be described as follows.



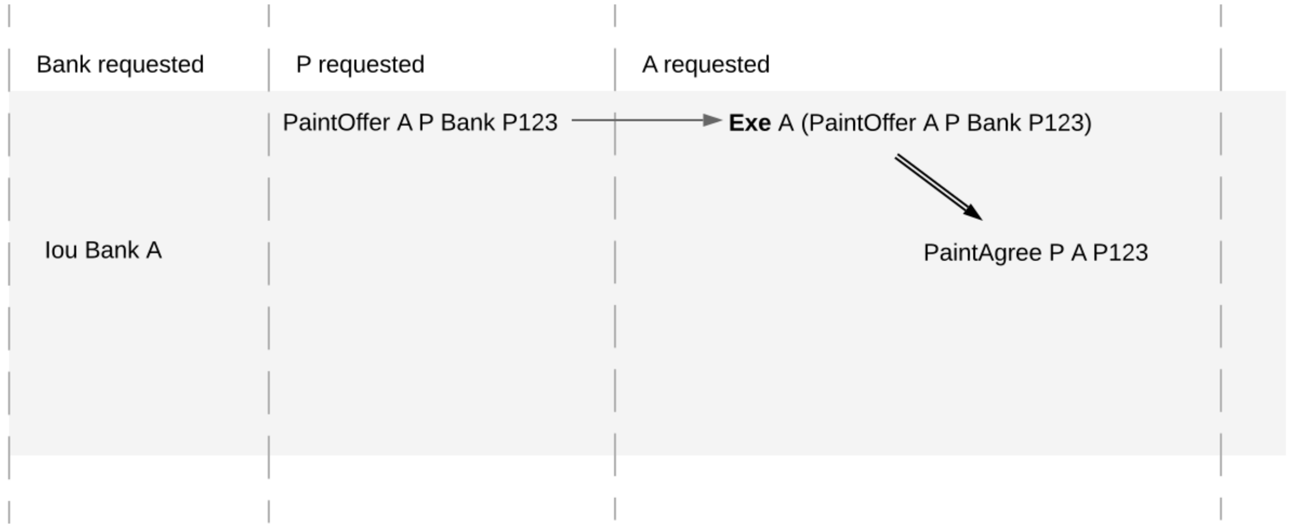
The boxes in the image are templates in the sense that the contract parameters in a box (such as obligor or owner) can be instantiated by arbitrary values of the appropriate type. To facilitate understanding, each box includes a label describing the intuitive purpose of the corresponding set of actions. 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 arbitrarily complex, and cannot conveniently be reproduced in this graphical representation. This is the role of DAML – it provides a much more convenient way of representing contract models. The link between DAML and contract models is explained in more detail in a *later section*.

To see the conformance criterion in action, assume that the contract model allows only the following actions on *PaintOffer* and *PaintAgree* contracts.


create:
PaintOffer houseOwner painter obligor refNo
create:
PaintAgree painter houseOwner refNo
accept:
Exe houseOwner (PaintOffer houseOwner painter obligor refNo)
Exe houseOwner (Iou obligor houseOwner)
PaintAgree painter houseOwner refNo
Iou obligor painter

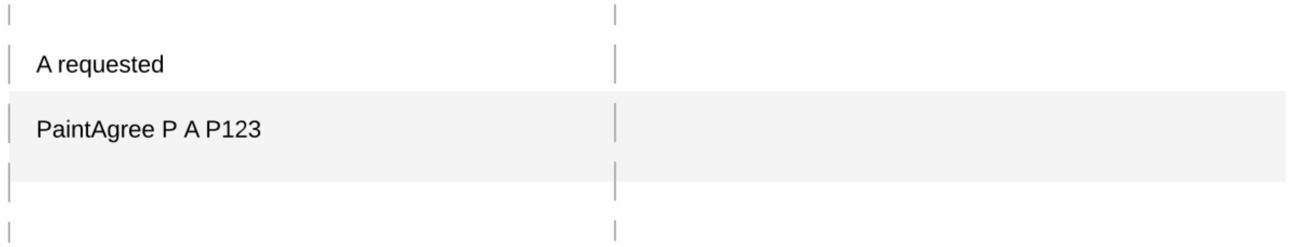
The problem with example where Alice changes the offer's outcome to avoid transferring the money now becomes apparent.



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.

6.2.2.4 Authorization

The last criterion rules out the last two problematic examples, *an obligation imposed on a painter*, and *the painter stealing Alice's money*. The first of those is visualized below.



The reason why the example is intuitively impermissible is that the *PaintAgree* contract is supposed to express that the painter has an obligation to paint Alice's house, but he never agreed to that obligation. On paper contracts, obligations are expressed in the body of the contract, and imposed on the contract's *signatories*.

Signatories, Agreements, and Maintainers

To capture these elements of real-world contracts, the **contract model** additionally specifies, for each contract in the system:

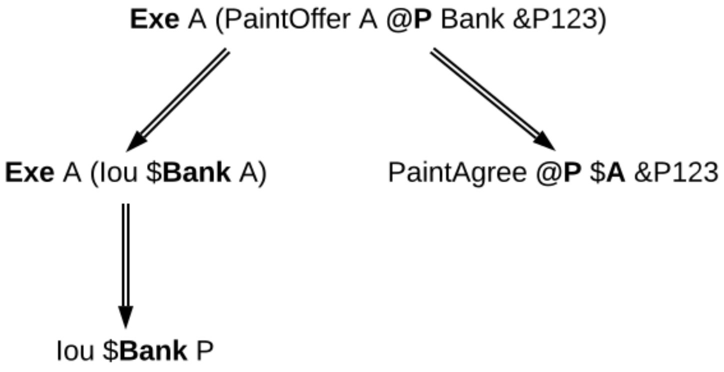
1. A non-empty set of **signatories**, the parties bound by the contract.
2. An optional **agreement text** associated with the contract, specifying the off-ledger, real-world obligations of the signatories.
3. If the contract is associated with a key, a non-empty set of **maintainers**, the parties that make sure that at most one unconsumed contract exists for the key. The maintainers must be a subset of the signatories and depend only on the key. This dependence is captured by the function *maintainers* that takes a key and returns the key's maintainers.

In the example, the contract model specifies that

1. an *Iou obligor owner* contract has only the *obligor* as a signatory, and no agreement text.

2. a *MustPay obligor owner* contract has both the *obligor* and the *owner* as signatories, with an agreement text requiring the obligor to pay the owner a certain amount, off the ledger.
3. a *PaintOffer houseOwner painter obligor refNo* contract has only the painter as the signatory, with no agreement text. Its associated key consists of the painter and the reference number. The painter is the maintainer.
4. a *PaintAgree houseOwner painter refNo* contract has both the house owner and the painter as signatories, with an agreement text requiring the painter to paint the house. The key consists of the painter and the reference number. The painter is the only maintainer.

In the graphical representation below, signatories of a contract are indicated with a dollar sign (as a mnemonic for an obligation) and use a bold font. Maintainers are marked with @ (as a mnemonic who enforces uniqueness). Since maintainers are always signatories, parties marked with @ are implicitly signatories. For example, annotating the paint offer acceptance action with signatories yields the image below.


Exe A (PaintOffer A @**P** Bank &P123)
Exe A (Iou $**Bank** A)
PaintAgree @**P** $**A** &P123
Iou $**Bank** P

Authorization Rules

Signatories allow one to precisely state that the painter has an obligation. The imposed obligation is intuitively invalid because the painter did not agree to this obligation. In other words, the painter did not *authorize* the creation of the obligation.

In a DAML ledger, a party can **authorize** a subaction of a commit in either of the following ways:

Every top-level action of the commit is authorized by all requesters of the commit.
Every consequence of an exercise action *act* on a contract *c* is authorized by all signatories of *c* and all actors of *act*.

The second authorization rule encodes the offer-acceptance pattern, which is a prerequisite for contract formation in contract law. The contract *c* is effectively an offer by its signatories who act as offerers. The exercise is an acceptance of the offer by the actors who are the offerees. The consequences of the exercise can be interpreted as the contract body so the authorization rules of 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:

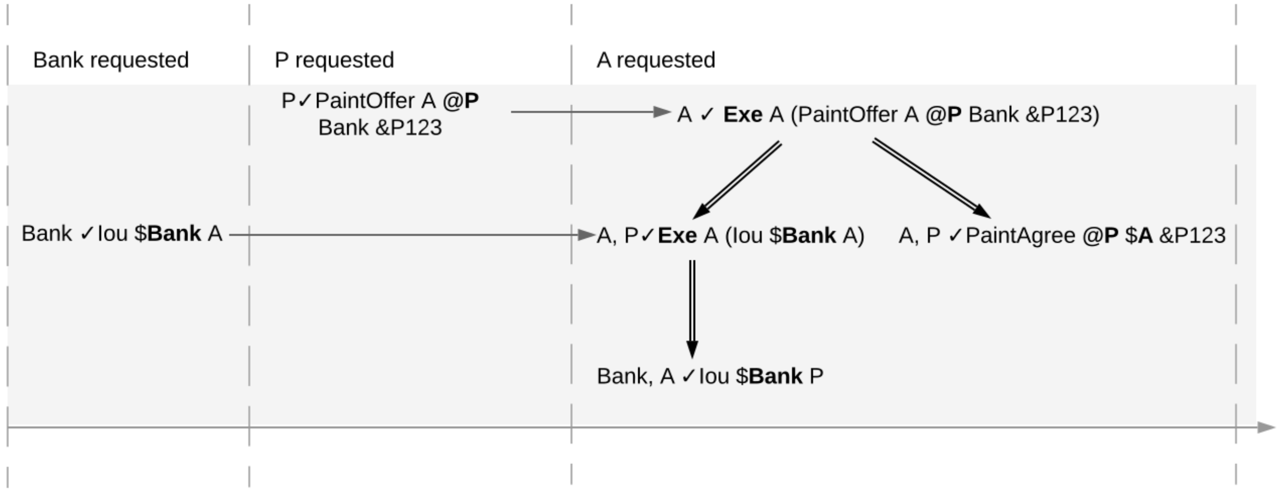
1. the required authorizers of a **Create** action on a contract *c* are the signatories of *c*.
2. the required authorizers of an **Exercise** or a **Fetch** action are its actors.
3. the required authorizers of a **NoSuchKey** assertion are the maintainers of the key.

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.

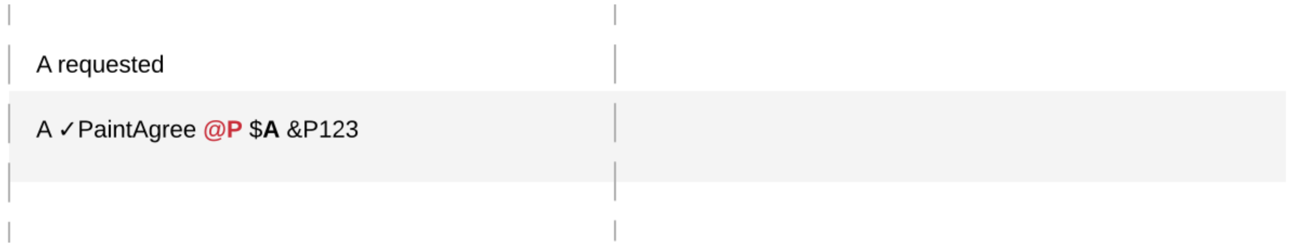
In the visualizations below, $\Pi \checkmark$ *act* denotes that the parties Π authorize the action *act*. The resulting authorizations are shown below.



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.

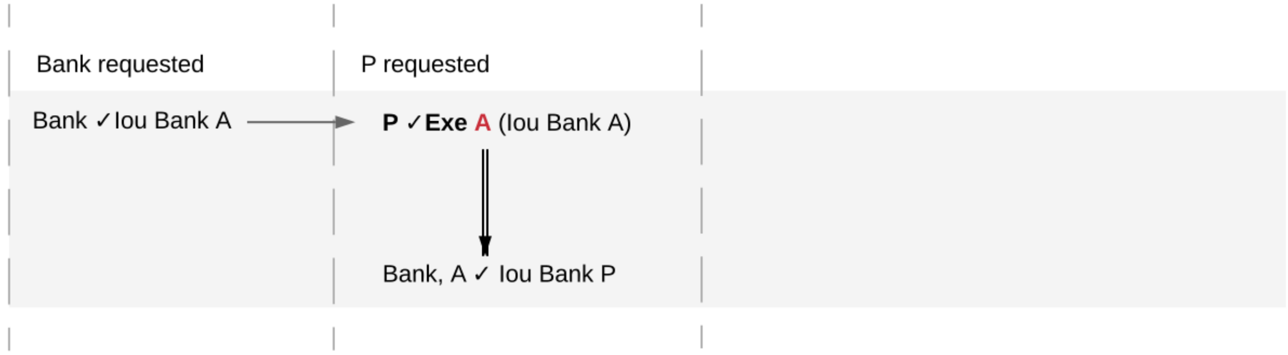
The third commit is more complicated. First, Alice authorizes the exercise on the paint offer by requesting it. She is the only actor on this exercise, so this complies with the authorization requirement. Since the painter is the signatory of the paint offer, and Alice the actor of the exercise, they 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 prohibited by our authorization criterion. In the first example, Alice forced the painter to paint her house. The authorizations for the example are shown below.



Alice authorizes the **Create** action on the *PaintAgree* contract by requesting it. However, the painter is also a signatory on the *PaintAgree* contract, but he did not authorize the **Create** action. Thus, this ledger is indeed not well-authorized.

In the second example, the painter steals money from Alice.



The bank authorizes the creation of the IOU by requesting this action. Similarly, the painter authorizes the exercise that transfers the IOU to him. However, the actor of this exercise is Alice, who has not authorized the exercise. Thus, this ledger is not well-authorized.

The rationale for making the maintainers required authorizers for a **NoSuchKey** assertion is discussed in the next section about *privacy*.

6.2.2.5 Valid Ledgers, Obligations, Offers and Rights

DAML ledgers are designed to mimic real-world interactions between parties, which are governed by contract law. The validity conditions on the ledgers, and the information contained in contract models have several subtle links to the concepts of the contract law that are worth pointing out.

First, in addition to the explicit off-ledger obligations specified in the agreement text, contracts also specify implicit **on-ledger obligations**, which result from consequences of the exercises on contracts. For example, the *PaintOffer* contains an on-ledger obligation for *A* to transfer her IOU in case she accepts the offer. Agreement texts are therefore only necessary to specify obligations that are not already modeled as permissible actions on the ledger. For example, *P*'s obligation to paint the house cannot be sensibly modeled on the ledger, and must thus be specified by the agreement text.

Second, every contract on a DAML ledger can simultaneously model both:

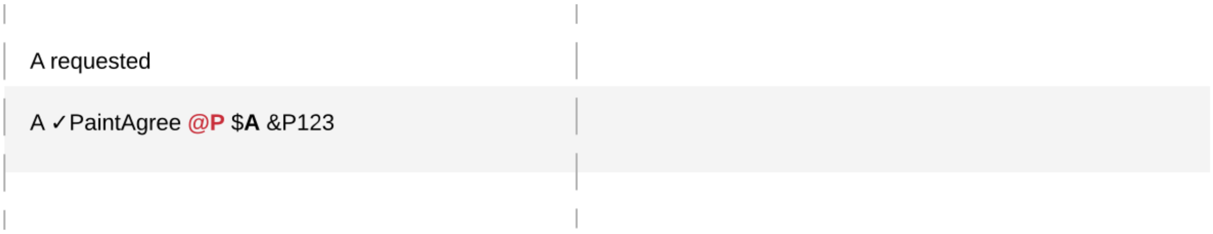
a real-world offer, whose consequences (both on- and off-ledger) are specified by the **Exercise** actions on the contract allowed by the contract model, and

a real-world contract proper , specified through the contract's (optional) agreement text.

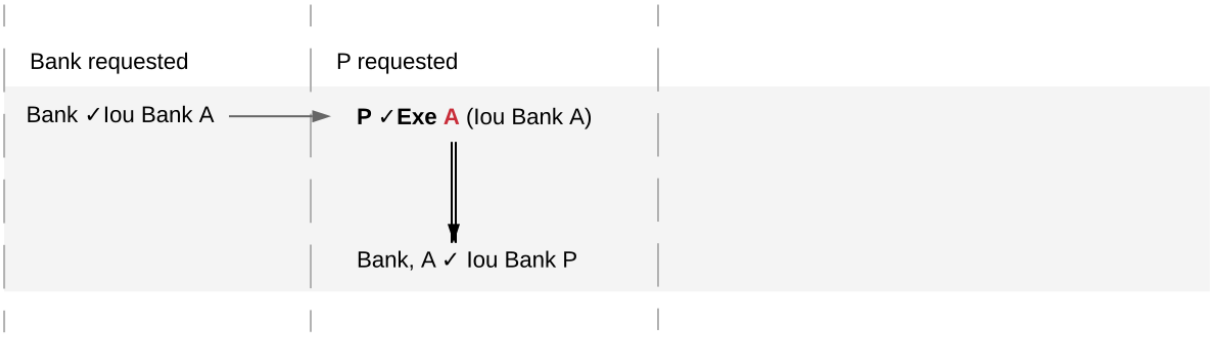
Third, in DAML ledgers, as in the real world, one person's rights are another person's obligations. For example, *A*'s right to accept the *PaintOffer* is *P*'s obligation to paint her house in case she accepts. In DAML ledgers, a party's rights according to a contract model are the exercise actions the party can perform according to the authorization and conformance rules.

Finally, validity conditions ensure three important properties of the DA ledger model, that mimic the contract law.

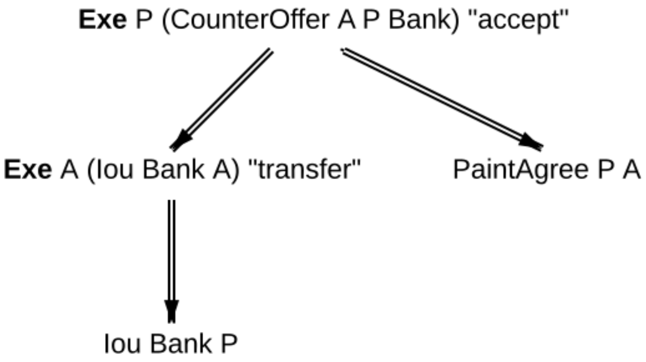
1. **Obligations need consent**. DAML ledgers follow the offer-acceptance pattern of the contract law, and thus ensures that all ledger contracts are formed voluntarily. For example, the following ledger is not valid.



2. **Consent is needed to take away on-ledger rights**. As only **Exercise** actions consume contracts, the rights cannot be taken away from the actors; the contract model specifies exactly who the actors are, and the authorization rules require them to approve the contract consumption. In the examples, Alice had the right to transfer her IOUs; painter's attempt to take that right away from her, by performing a transfer himself, was not valid.



Parties can still **delegate** their rights to other parties. For example, assume that Alice, instead of accepting painter's offer, decides to make him a counteroffer instead. The painter can then accept this counteroffer, with the consequences as before:


Exe P (CounterOffer A P Bank) "accept"
Exe A (Iou Bank A) "transfer"
PaintAgree P A
Iou Bank P

Here, by creating the *CounterOffer* contract, Alice delegates her right to transfer the IOU contract to the painter. In case of delegation, prior to submission, the requester must get informed about the contracts that are part of the requested transaction, but where the requester is not a signatory. In the example above, the painter must learn about the existence of the IOU for Alice before he can request the acceptance of the *CounterOffer*. The concepts of observers and divulgence, introduced in the next section, enable such scenarios.

3. **On-ledger obligations cannot be unilaterally escaped**. Once an obligation is recorded on a DAML ledger, it can only be removed in accordance with the contract model. For example, assuming the IOU contract model shown earlier, if the ledger records the creation of a *MustPay* contract, the bank cannot later simply record an action that consumes this contract:

. . .
Bank commits
. . .
Exe Bank (MustPay Bank A)
MustPay Bank A

That is, this ledger is invalid, as the action above is not conformant to the contract model.

6.2.3 Privacy

The previous sections have addressed two out of three questions posed in the introduction: what the ledger looks like , and who may request which changes . This section addresses the last one, who sees which changes and data . That is, it explains the privacy model for DAML ledgers.

The privacy model of DAML Ledgers is based on a **need-to-know basis**, and provides privacy **on the level of subtransactions**. Namely, a party learns only those parts of ledger changes that affect contracts in which the party has a stake, and the consequences of those changes. And maintainers see all changes to the contract keys they maintain.

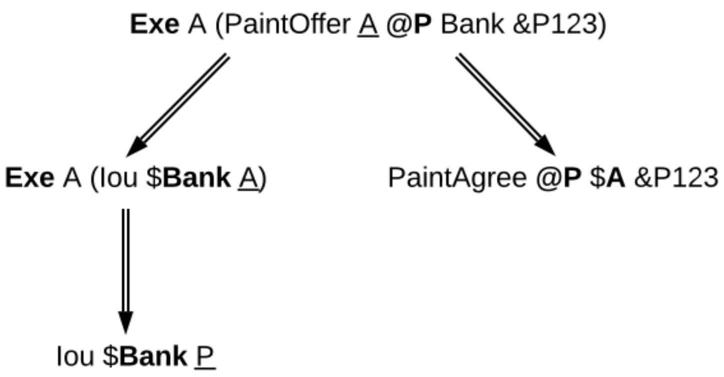
To make this more precise, a stakeholder concept is needed.

6.2.3.1 Contract Observers and Stakeholders

Intuitively, as signatories are bound by a contract, they have a stake in it. Actors might not be bound by the contract, but they still have a stake in their actions, as these are the actor's rights. Generalizing this, **observers** are parties who might not be bound by the contract, but still have the right to see the contract. For example, Alice should be an observer of the *PaintOffer*, such that she is made aware that the offer exists.

Signatories are already determined by the contract model discussed so far. The full **contract model** additionally specifies the 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.

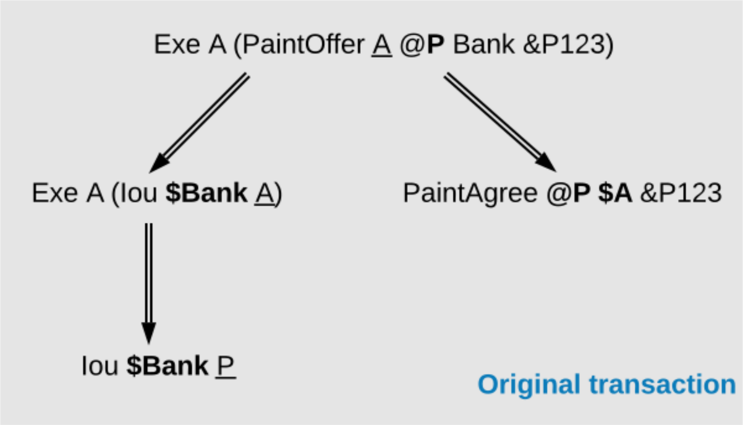
In the graphical representation of the paint offer acceptance below, observers who are not signatories are indicated by an underline.

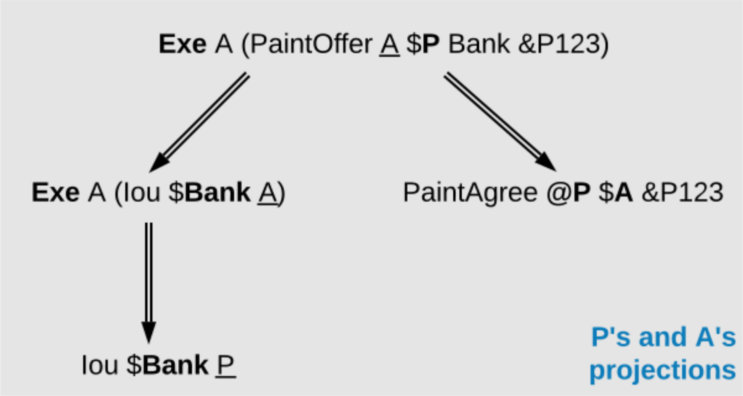


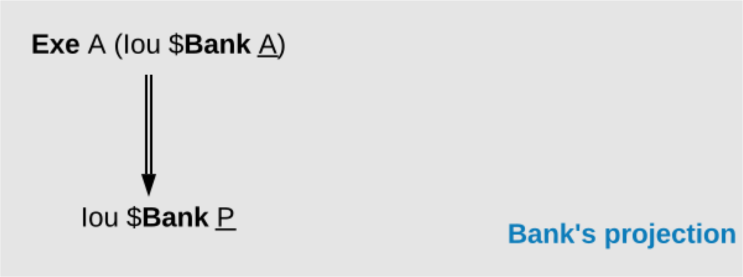
6.2.3.2 Projections

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 through *projections* of a transaction, which define the view that each party gets on a transaction. Intuitively, given a transaction within a commit, a party will see only the subtransaction consisting of all actions on contracts where the party is a stakeholder. Thus, privacy is obtained on the subtransaction level.

An example is given below. The transaction that consists only of Alice's acceptance of the *PaintOffer* is projected for each of the three parties in the example: the painter, Alice, and the bank.







Since both the painter and Alice are stakeholders of the *PaintOffer* contract, the exercise on this contract is kept in the projection of both parties. Recall that consequences of an exercise action are a part of the action. Thus, both parties also see the exercise on the *Iou Bank A* contract, and the creations of the *Iou Bank P* and *PaintAgree* contracts.

The bank is *not* a stakeholder on the *PaintOffer* contract (even though it is mentioned in the contract).

Thus, the projection for the bank is obtained by projecting the consequences of the exercise on the *PaintOffer*. The bank is a stakeholder in the contract *Iou Bank A*, so the exercise on this contract is kept in the bank's projection. Lastly, as the bank is not a stakeholder of the *PaintAgree* contract, the corresponding **Create** action is dropped from the bank's projection.

Note the privacy implications of the bank's projection. While the bank learns that a transfer has occurred from *A* to *P*, the bank does *not* learn anything about *why* the transfer occurred. In practice, this means that the bank does not learn what *A* is paying for, providing privacy to *A* and *P* with respect to the bank.

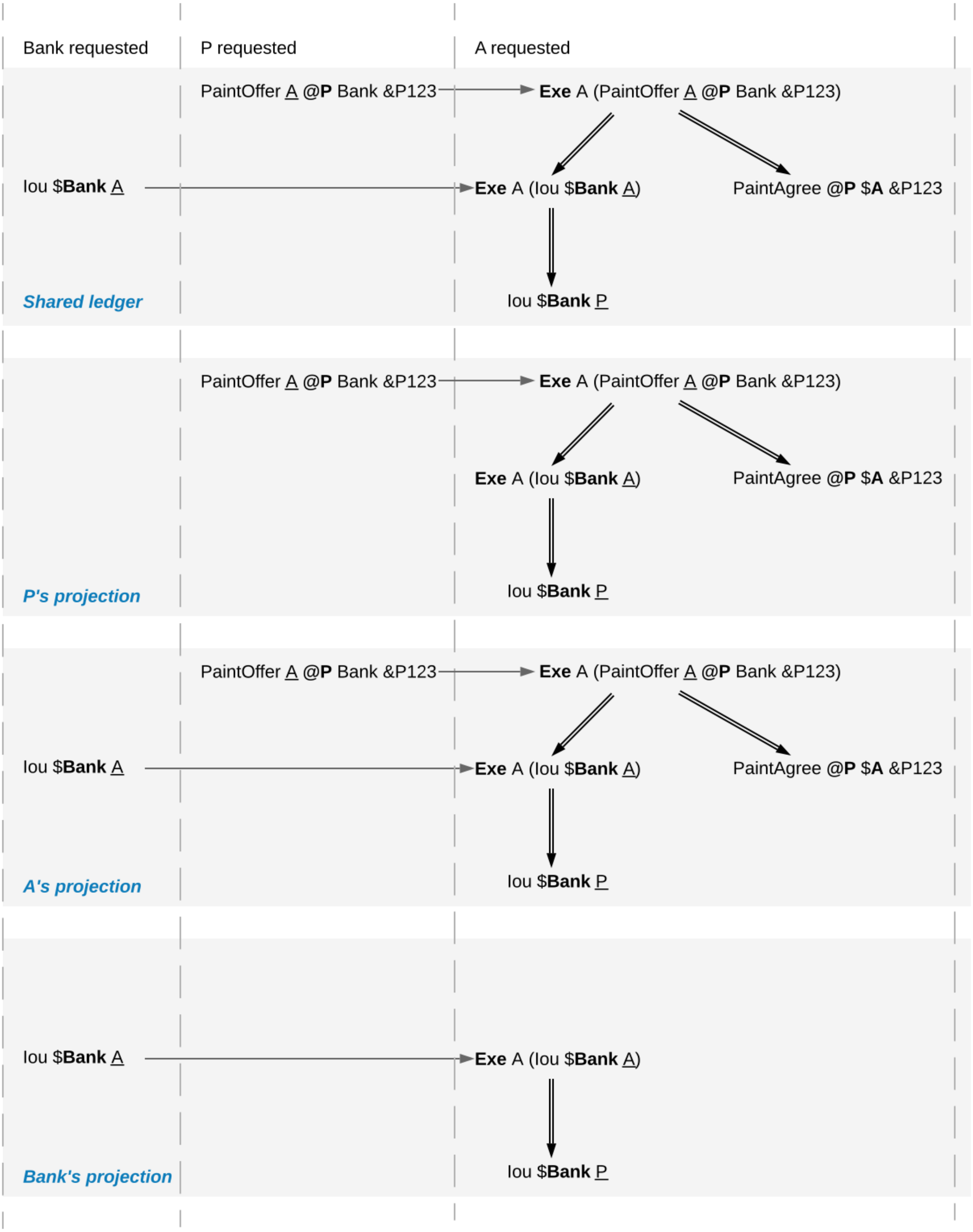
As a design choice, DAML Ledgers show to observers on a contract only the *state changing* actions on the contract. More precisely, **Fetch** and non-consuming **Exercise** actions are not shown to the observers - except when they are the actors of these actions. This motivates the following definition: 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 subtransaction obtained by doing the following for each action act_i:

1. If *p* is an informee of act_i, keep act_i as-is.
2. Else, if act_i has consequences, replace act_i by the projection (for *p*) of its consequences, which might be empty.
3. Else, drop act_i.

Finally, the **projection of a ledger** *l* for a party *p* is a list of transactions obtained by first projecting the transaction of each commit in *l* for *p*, and then removing all empty transactions from the result. Note that the projection of a ledger is not a ledger, but a list of transactions. Projecting the ledger of our complete paint offer example yields the following projections for each party:



Examine each party's projection in turn:

1. The painter does not see any part of the first commit, as he is not a stakeholder of the *Iou Bank A* contract. Thus, this transaction is not present in the projection for the painter at all. However, the painter is a stakeholder in the *PaintOffer*, so he sees both the creation and the exercise of this contract (again, recall that all consequences of an exercise action are a part of the action

itself).

2. Alice is a stakeholder in both the *Iou Bank A* and *PaintOffer A B Bank* contracts. As all top-level actions in the ledger are performed on one of these two contracts, Alice's projection includes all the transactions from the ledger intact.
3. The Bank is only a stakeholder of the IOU contracts. Thus, the bank sees the first commit's transaction as-is. The second commit's transaction is, however dropped from the bank's projection. The projection of the last commit's transaction is as described above.

Ledger projections do not always satisfy the definition of consistency, even if the ledger does. For example, in P's view, *Iou Bank A* is exercised without ever being created, and thus without being made active. Furthermore, projections can in general be non-conformant. However, the projection for a party *p* is always

internally consistent for all contracts,
consistent for all contracts on which *p* is a stakeholder, and
consistent for the keys that *p* is a maintainer of.

In other words, *p* is never a stakeholder on any input contracts of its projection. Furthermore, if the contract model is **subaction-closed**, which means that for every action *act* in the model, all subactions of *act* are also in the model, then the projection is guaranteed to be conformant. As we will see shortly, DAML-based contract models are conformant. Lastly, as projections carry no information about the requesters, we cannot talk about authorization on the level of projections.

6.2.3.3 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 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.

6.2.3.4 Divulgence: When Non-Stakeholders See Contracts

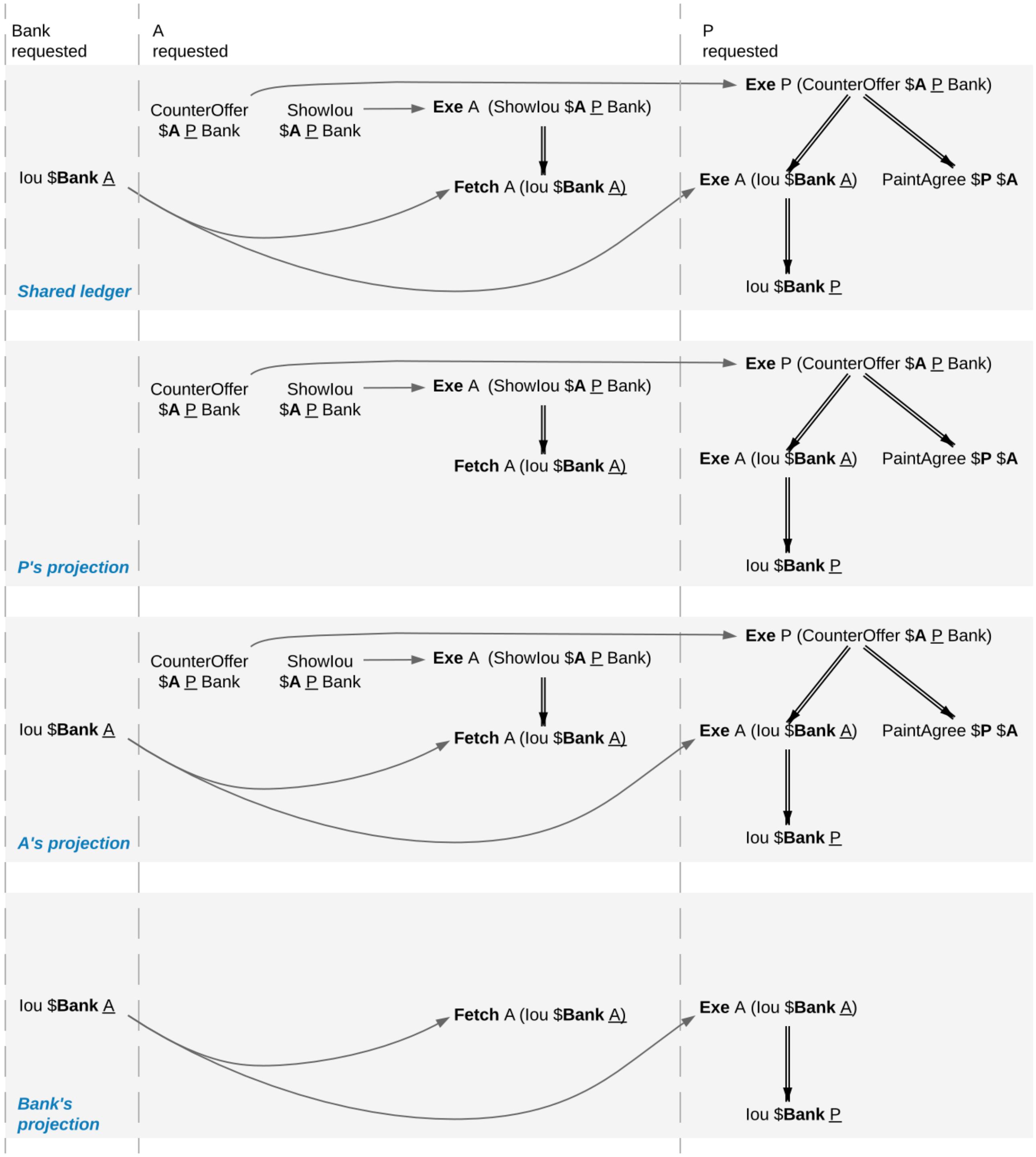
The guiding principle for the privacy model of DAML ledgers is that contracts should only be shown to their stakeholders. However, ledger projections can cause contracts to become visible to other parties as well.

In the example of *ledger projections of the paint offer*, the exercise on the *PaintOffer* is visible to both the painter and Alice. As a consequence, the exercise on the *Iou Bank A* is visible to the painter, and the creation of *Iou Bank P* is visible to Alice. As actions also contain the contracts they act on, *Iou Bank A* was thus shown to the painter and *Iou Bank P* was shown to Alice.

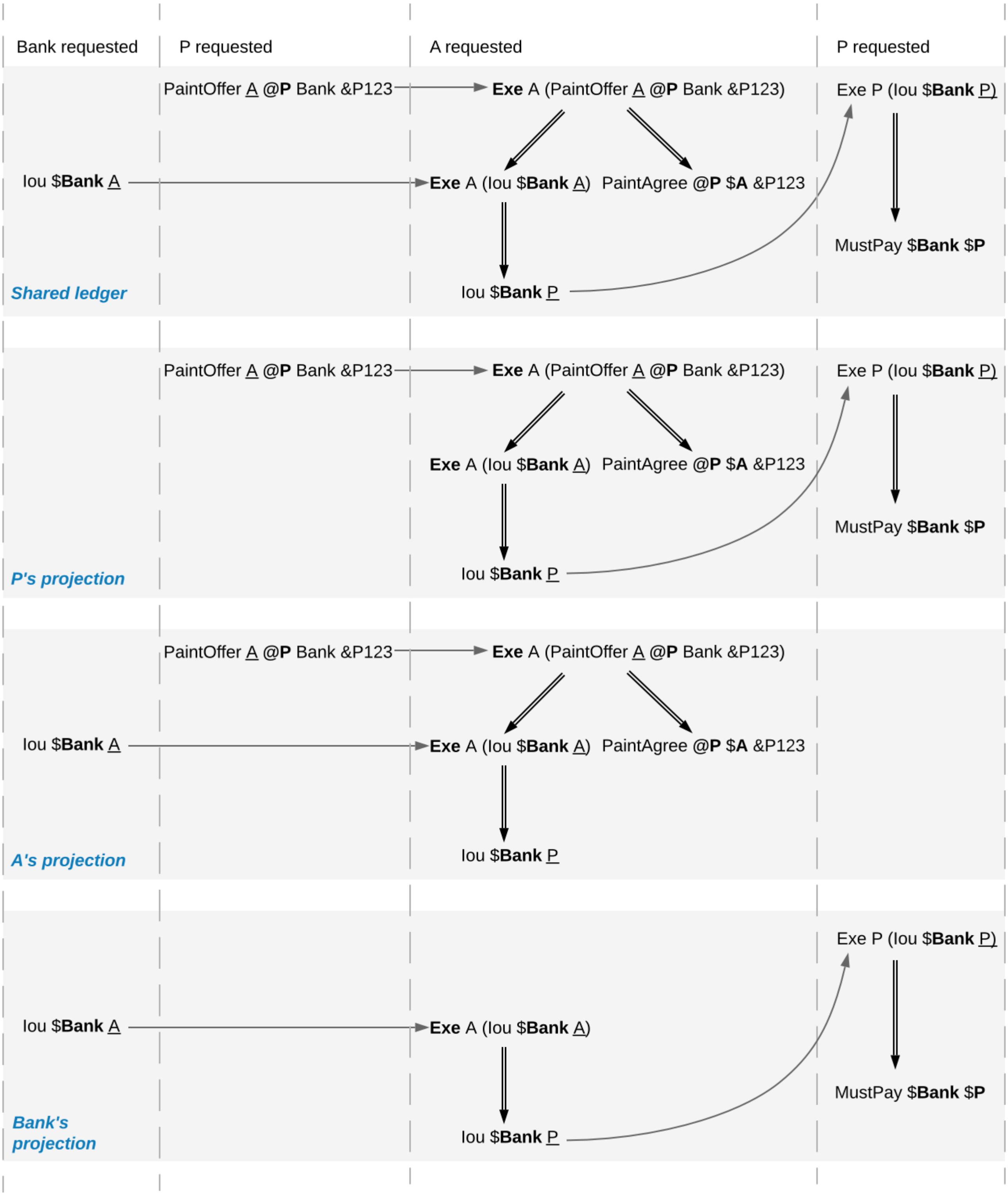
Showing contracts to non-stakeholders through ledger projections is called **divulgence**. Divulgence is a deliberate choice in the design of DAML ledgers. In the paint offer example, the only proper way to accept the offer is to transfer the money from Alice to the painter. Conceptually, at the instant where the offer is accepted, its stakeholders also gain a temporary stake in the actions on the two *Iou* contracts, even though they are never recorded as stakeholders in the contract model. Thus, they are allowed to see these actions through the projections.

More precisely, every action *act* on *c* is shown to all informees of all ancestor actions of *act*. These informees are called the **witnesses** of *act*. If one of the witnesses *W* is not a stakeholder on *c*, then *act* and *c* are said to be **divulged** to *W*. Note that only **Exercise** actions can be ancestors of other actions.

Divulgence can be used to enable delegation. For example, consider the scenario where Alice makes a counteroffer to the painter. Painter's acceptance entails transferring the IOU to him. To be able to construct the acceptance transaction, the painter first needs to learn about the details of the IOU that will be transferred to him. To give him these details, Alice can fetch the IOU in a context visible to the painter:



In the example, the context is provided by consuming a *ShowIou* contract on which the painter is a stakeholder. This now requires an additional contract type, compared to the original paint offer example. An alternative approach to enable this workflow, without increasing the number of contracts required, is to replace the original *Iou* contract by one on which the painter is 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.

6.2.4 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 templates such that the template arguments satisfy the *ensure* clause of the template
2. **Exercise** actions on a contract instance corresponding to choices on that template, with given choice arguments, such that:
   1. The actors match the controllers of the choice. That is, the controllers define the *required authorizers* of the choice.
   2. The 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 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 stakeholders of *c2*.

   A *fetchByKey* statement also produces a **Fetch** action with the actors determined in the same way. A *lookupByKey* statement that finds a contract also translates into a **Fetch** action, but all maintainers of the key are the actors.
4. **NoSuchKey** assertions corresponding to a *lookupByKey* update statement for the given key that does not find a contract.

An instance of a 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 template exactly describes the contract model of a simple IOU with a unit amount, shown earlier.

```
template MustPay with
    obligor : Party
    owner : Party
  where
    signatory obligor, owner
    agreement
      show obligor <> " must pay " <>
      show owner <> " one unit of value"

template Iou with
    obligor : Party
    owner : Party
  where
    signatory obligor

    controller owner can
      Transfer
        : ContractId Iou
        with newOwner : Party
        do create Iou with obligor; owner = newOwner

    controller owner can
      Settle
        : ContractId MustPay
        do create MustPay with obligor; owner
```

In this example, the owner is automatically made an observer on the contract, as the `Transfer` and `Settle` choices use the `controller owner can` syntax.

The template identifiers of contracts are created through a content-addressing scheme. This means every contract is self-describing in a sense: it constrains its stakeholder annotations and all DAML-conformant actions on itself. As a consequence, one can talk about the DAML contract model, as a single contract model encoding all possible instances of all possible templates. This model is subaction-closed; all exercise and create actions done within an update block are also always permissible as top-level actions.

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
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 Node.js bindings

Chapter 8

Early Access Features

8.1 Navigator Console

8.1.1 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:

8.1.1.1 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.

8.1.1.2 Example query using plain SQL

Filter on the template id of contracts:

```
sql select count (*) from contract where template_id like '%Offer%'
```

8.1.1.3 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 DAML 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:

8.1.2 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
```

(continues on next page)

(continued from previous page)

```
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.

8.1.2.1 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.
```

8.1.2.2 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
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
```

8.1.2.3 Exiting Navigator Console

To exit Navigator Console, use the `quit` command:

```
>quit
Bye.
```

8.1.2.4 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.

8.1.3 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
    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
```

(continues on next page)

(continued from previous page)

```
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
```

8.1.4 Choosing a party

Privacy is an important aspect of a DAML 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.)

8.1.5 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 all DAML Ledgers.)

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
```

8.1.6 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
```

8.1.7 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
```

(continues on next page)

(continued from previous page)

```
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:
```

8.1.8 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║
```

(continues on next page)

(continued from previous page)

```
╠═══════╪═══════════════════╪══════════════════════════╣
║#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.

8.1.9 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
        expirationDate: 2020-01-01T00:00:00Z
Status:
```

(continues on next page)

(continued from previous page)

```
    Status: Success
    TransactionId: 2005
```

8.1.10 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
```

8.1.10.1 Advanced usage

8.1.11 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> {
        party = <PARTYNAME>
    }
```

(continues on next page)

(continued from previous page)

```
    ..
}
```

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" }
}
```

8.1.12 Using Navigator with DAML Ledgers

By default, Navigator is configured to use an unencrypted connection to the ledger.

To run Navigator against a secured DAML Ledger, configure TLS certificates using the `--pem`, `--crt`, and `--cacrt` command line parameters.

Details of these parameters are explained in the command line help:

```
daml navigator --help
```

8.2 Extractor

8.2.1 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)

8.2.2 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
```

8.2.3 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.

8.2.4 Running the Extractor

The basic command to run the Extractor is:

```
$ daml extractor [options]
```

For what options to use, see the next sections.

8.2.5 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.

8.2.6 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]
```

8.2.7 Authorize Extractor

If you are running Extractor against a Ledger API server that verifies authorization, you must provide the access token when you start it.

The access token retrieval depends on the specific DAML setup you are working with: please refer to the ledger operator to learn how.

Once you have retrieved your access token, you can provide it to Extractor by storing it in a file and provide the path to it using the `--access-token-file` command line option.

Both in the case in which the token cannot be read from the provided path or if the Ledger API reports an authorization error (for example due to token expiration), Extractor will keep trying to read and use it and report the error via logging. This retry mechanism allows expired token to be overwritten with valid ones and keep Extractor going from where it left off.

8.2.8 Full list of options

To see the full list of options, run the `--help` command, which gives the following output:

```
Usage: extractor [prettyprint|postgresql] [options]

Command: prettyprint [options]
Pretty print contract template and transaction data to stdout.
  --width <value>           How wide to allow a pretty-printed value to
↪become before wrapping.
                            Optional, default is 120.
  --height <value>          How tall to allow each pretty-printed output to
↪become before
                            it is truncated with a `...`.
                            Optional, default is 1000.

Command: postgresql [options]
Extract data into a PostgreSQL database.
  --connecturl <value>      Connection url for the `org.postgresql.Driver`
↪driver. For examples,
                            visit https://jdbc.postgresql.org/documentation/
↪80/connect.html
```

(continues on next page)

(continued from previous page)

```
  --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.
  --ledger-api-inbound-message-size-max <bytes>
                           Maximum message size from the ledger API.
↪Default is 52428800 (50MiB).
  --party <value>          The party or parties whose contract data should
↪be extracted.
                         Specify multiple parties separated by a comma, e.
↪g. Foo,Bar
  -t, --templates <module1>:<entity1>,<module2>:<entity2>...
                           The list of templates to subscribe for.
↪Optional, defaults to all ledger templates.
  --from <value>           The transaction offset (exclusive) for the
↪snapshot start position.
                           Must not be greater than the current latest
↪transaction offset.
                           Optional, defaults to the beginning of the
↪ledger.
                           Currently, only the integer-based Sandbox
↪offsets are supported.
  --to <value>             The transaction offset (inclusive) for the
↪snapshot end position.
                           Use "head" to use the latest transaction offset
↪at the time
                           the extraction first started, or "follow" to
↪stream indefinitely.
                           Must not be greater than the current latest
↪offset.
                           Optional, defaults to "follow".
  --help                   Prints this usage text.

TLS configuration:
  --pem <value>            TLS: The pem file to be used as the private key.
  --crt <value>            TLS: The crt file to be used as the cert chain.
                           Required if any other TLS parameters are set.
  --cacrt <value>          TLS: The crt file to be used as the trusted
↪root CA.

Authorization:
  --access-token-file <value>
                           provide the path from which the access token
↪will be read, required if the Ledger API server verifies authorization,
↪no default
```

Some options are tied to a specific subcommand, like `--connecturl` only makes sense for the `postgresql`, while others are general, like `--party`.

8.2.9 Output format

To understand the format that Extractor outputs into a PostgreSQL database, you need to understand how the ledger stores data.

The DAML Ledger is composed of transactions, which contain events. Events can represent:

- creation of contracts (create event), or
- exercise of a choice on a contract (exercise event).

A contract on the ledger is either active (created, but not yet archived), or archived. The relationships between transactions and contracts are captured in the database: all contracts have pointers (foreign keys) to the transaction in which they were created, and archived contracts have pointers to the transaction in which they were archived.

8.2.10 Transactions

Transactions are stored in the `transaction table` in the `public` schema, with the following structure

```
CREATE TABLE transaction
  (transaction_id TEXT PRIMARY KEY NOT NULL
  ,seq BIGSERIAL UNIQUE NOT NULL
  ,workflow_id TEXT
  ,effective_at TIMESTAMP NOT NULL
  ,extracted_at TIMESTAMP DEFAULT NOW()
  ,ledger_offset TEXT NOT NULL
  );
```

- **transaction_id**: The transaction ID, as appears on the ledger. This is the primary key of the table.
- **transaction_id, effective_at, workflow_id, ledger_offset**: These columns are the properties of the transaction on the ledger. For more information, see the specification.
- **seq**: Transaction IDs should be treated as arbitrary text values: you can't rely on them for ordering transactions in the database. However, transactions appear on the Ledger API transaction stream in the same order as they were accepted on the ledger. You can use this to work around the arbitrary nature of the transaction IDs, which is the purpose of the `seq` field: it gives you a total ordering of the transactions, as they happened from the perspective of the ledger. Be aware that `seq` is not the exact index of the given transaction on the ledger. Due to the privacy model of DAML Ledgers, the transaction stream won't deliver a transaction which doesn't concern the party which is subscribed. The transaction with `seq` of 100 might be the 1000th transaction on the ledger; in the other 900, the transactions contained only events which mustn't be seen by you.
- **extracted_at**: The `extracted_at` field means the date the transaction row and its events were inserted into the database. When extracting historical data, this field will point to a possibly much later time than `effective_at`.

8.2.11 Contracts

Create events and contracts that are created in those events are stored in the `contract` table in the `public` schema, with the following structure

```
CREATE TABLE contract
  (event_id TEXT PRIMARY KEY NOT NULL
  ,archived_by_event_id TEXT DEFAULT NULL
  ,contract_id TEXT NOT NULL
  ,transaction_id TEXT NOT NULL
  ,archived_by_transaction_id TEXT DEFAULT NULL
  ,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*.

8.2.12 Exercises

Exercise events are stored in the `exercise` table in the `public` schema, with the following structure:

```
CREATE TABLE
  exercise
  (event_id TEXT PRIMARY KEY NOT NULL
  ,transaction_id TEXT NOT NULL
  ,is_root_event BOOLEAN NOT NULL
  ,contract_id TEXT NOT NULL
  ,package_id TEXT NOT NULL
  ,template TEXT NOT NULL
  ,contract_creating_event_id TEXT NOT NULL
  ,choice TEXT NOT NULL
  ,choice_argument JSONB NOT NULL
  ,acting_parties JSONB NOT NULL
  ,consuming BOOLEAN NOT NULL
  ,witness_parties JSONB NOT NULL
  ,child_event_ids JSONB NOT NULL
  );
```

package_id, template: The fields `package_id` and `template` are the exploded version of the `template_id` property of the ledger event.

is_root_event: Indicates whether the event in which the contract was created was a root event of the corresponding transaction.

transaction_id: The transaction_id field refers to the transaction in which the contract was created.

The other columns are properties of the ExercisedEvent class in a transaction. For more information, see the specification.

8.2.13 JSON format

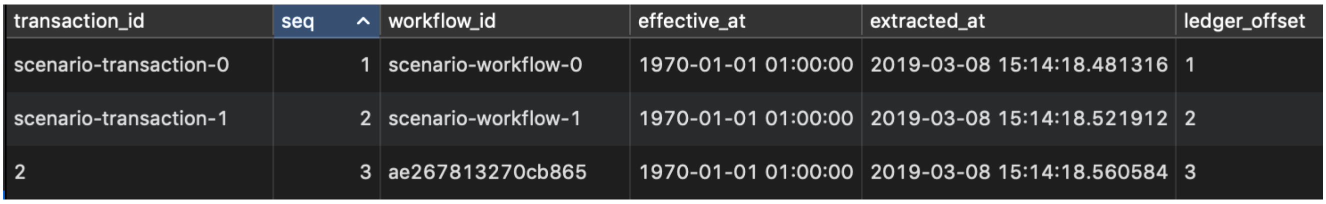
Extractor stores create and choice arguments using the *DAML-LF JSON Encoding*. The parameters of the JSON schema are instantiated as follows in Extractor:

```
encodeDecimalAsString: true
encodeInt64AsString: false
```

8.2.14 Examples of output

The following examples show you what output you should expect. The Sandbox has already run the scenarios of a DAML model that created two transactions: one creating a Main:RightOfUseOffer and one accepting it, thus archiving the original contract and creating a new Main:RightOfUseAgreement contract. We also added a new offer manually.

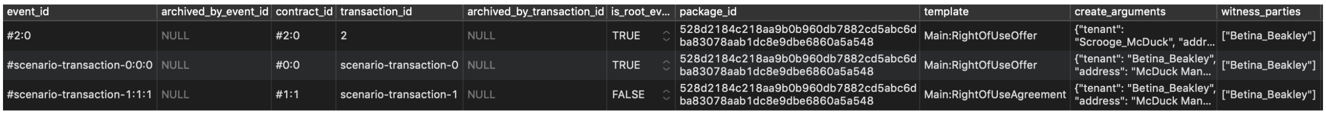
This is how the transaction table looks after extracting data from the ledger:



transaction_id	seq	workflow_id	effective_at	extracted_at	ledger_offset
scenario-transaction-0	1	scenario-workflow-0	1970-01-01 01:00:00	2019-03-08 15:14:18.481316	1
scenario-transaction-1	2	scenario-workflow-1	1970-01-01 01:00:00	2019-03-08 15:14:18.521912	2
2	3	ae267813270cb865	1970-01-01 01:00:00	2019-03-08 15:14:18.560584	3

You can see that the transactions which were part of the scenarios have the format scenario-transaction-{n}, while the transaction created manually is a simple number. This is why the seq field is needed for ordering. In this output, the ledger_offset field has the same values as the seq field, but you should expect similarly arbitrary values there as for transaction IDs, so better rely on the seq field for ordering.

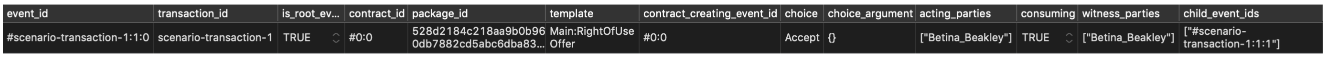
This is how the contract table looks:



event_id	archived_by_event_id	contract_id	transaction_id	archived_by_transaction_id	is_root_ev...	package_id	template	create_arguments	witness_parties
#2:0	NULL	#2:0	2	NULL	TRUE	528d2184c218aa9b0b960db7882cd5abc6d ba83078aab1dc8e9dbe6860a5a548	Main:RightOfUseOffer	{"tenant": "Scrooge_McDuck", "addr...	["Betina_Beakley"]
#scenario-transaction-0:0:0	NULL	#0:0	scenario-transaction-0	NULL	TRUE	528d2184c218aa9b0b960db7882cd5abc6d ba83078aab1dc8e9dbe6860a5a548	Main:RightOfUseOffer	{"tenant": "Betina_Beakley", "address": "McDuck Man...	["Betina_Beakley"]
#scenario-transaction-1:1:1	NULL	#1:1	scenario-transaction-1	NULL	FALSE	528d2184c218aa9b0b960db7882cd5abc6d ba83078aab1dc8e9dbe6860a5a548	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	528d2184c218aa9b0b96 0db7882cd5abc6dba83...	Main:RightOfUse Offer	#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.

8.2.15 Dealing with schema evolution

When updating packages, you can end up with multiple versions of the same package in the system.

Let's say you have a template called `My.Company.Finance.Account`:

```
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:

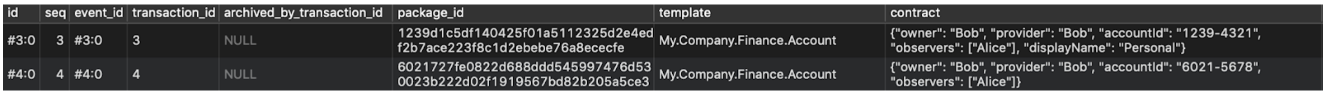
```
{
  "owner":"Bob",
  "provider":"Bob",
```

(continues on next page)

(continued from previous page)

```
  "accountId":"1239-4321",
  "observers":[
      "Alice"
  ],
  "displayName":"Personal"
}
```

They will look like this when extracted:

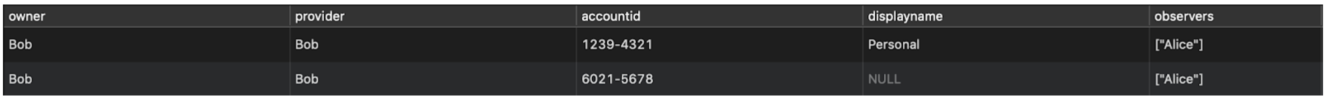


id	seq	event_id	transaction_id	archived_by_transaction_id	package_id	template	contract
#3:0	3	#3:0	3	NULL	1239d1c5df140425f01a5112325d2e4ed f2b7ace223f8c1d2ebebe76a8ececfe	My.Company.Finance.Account	{"owner": "Bob", "provider": "Bob", "accountId": "1239-4321", "observers": ["Alice"], "displayName": "Personal"}
#4:0	4	#4:0	4	NULL	6021727fe0822d688ddd545997476d53 0023b222d02f1919567bd82b205a5ce3	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"]

8.2.16 Logging

By default, the Extractor logs to stderr, with INFO verbose level. To change the level, use the `-DLOGLEVEL=[level]` option, e.g. `-DLOGLEVEL=TRACE`.

You can supply your own logback configuration file via the standard method: https://logback.qos.ch/manual/configuration.html

8.2.17 Continuity

When you terminate the Extractor and restart it, it will continue from where it left off. This happens because, when running, it saves its state into the `state` table in the `public` schema of the database. When started, it reads the contents of this table. If there's a saved state from a previous run, it restarts from where it left off. There's no need to explicitly specify anything, this is done automatically.

DO NOT modify content of the `state` table. Doing so can result in the Extractor not being able to continue running against the database. If that happens, you must delete all data from the database and start again.

If you try to restart the Extractor against the same database but with different configuration, you will get an error message indicating which parameter is incompatible with the already exported data. This happens when the settings are incompatible: for example, if previously contract data for the party `Alice` was extracted, and now you want to extract for the party `Bob`.

The only parameters that you can change between two sessions running against the same database are the connection parameters to both the ledger and the database. Both could have moved to different addresses, and the fact that it's still the same Ledger will be validated by using the Ledger ID (which is saved when the Extractor started its work the first time).

8.2.18 Fault tolerance

Once the Extractor connects to the Ledger Node and the database and creates the table structure from the fetched DAML packages, it wraps the transaction stream in a restart logic with an exponential backoff. This results in the Extractor not terminating even when the transaction stream is aborted for some reason (the ledger node is down, there's a network partition, etc.).

Once the connection is back, it continues the stream from where it left off. If it can't reach the node on the host/port pair the Extractor was started with, you need to manually stop it and restart with the updated address.

Transactions on the ledger are inserted into PostgreSQL as atomic SQL transactions. This means either the whole transaction is inserted or nothing, so you can't end up with inconsistent data in the database.

8.2.19 Troubleshooting

8.2.19.1 Can't connect to the Ledger Node

If the Extractor can't connect to the Ledger node on startup, you'll see a message like this in the logs, and the Extractor will terminate:

```
16:47:51.208 ERROR c.d.e.Main$@[akka.actor.default-dispatcher-7] - FAILURE:
io.grpc.StatusRuntimeException: UNAVAILABLE: io exception.
Exiting...
```

To fix this, make sure the Ledger node is available from where you're running the Extractor.

8.2.19.2 Can't connect to the database

If the database isn't available before the transaction stream is started, the Extractor will terminate, and you'll see the error from the JDBC driver in the logs:

```
17:19:12.071 ERROR c.d.e.Main$@[kka.actor.default-dispatcher-5] - FAILURE:
org.postgresql.util.PSQLException: FATAL: database "192.153.1.23:daml_export" does not exist.
Exiting...
```

To fix this, make sure make sure the database exists and is available from where you're running the Extractor, the username and password your using are correct, and you have the credentials to connect to the database from the network address where the you're running the Extractor.

If the database connection is broken while the transaction stream was already running, you'll see a similar message in the logs, but in this case it will be repeated: as explained in the *Fault tolerance* section, the transaction stream will be restarted with an exponential backoff, giving the database, network or any other trouble resource to get back into shape. Once everything's back in order, the stream will continue without any need for manual intervention.

8.3 DAML Integration Kit - ALPHA

8.3.1 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*.

8.3.1.1 Downloading the tool

Download the Ledger API Test Tool from Maven and save it as `ledger-api-test-tool.jar` in your current directory.

8.3.1.2 Running the tool against a custom Ledger API endpoint

Run this command to test your Ledger API endpoint exposed at host `<host>` and at a port `<port>`:

```
$ java -jar ledger-api-test-tool.jar <host>:<port>
```

For example:

```
$ java -jar ledger-api-test-tool.jar localhost:6865
```

The tool will upload the required DARs to the ledger, and then run all tests.

If any test embedded in the tool fails, it will print out details of the failure for further debugging.

8.3.1.3 Exploring options the tool provides

Run the tool with `--help` flag to obtain the list of options the tool provides:

```
$ java -jar ledger-api-test-tool.jar --help
```

Selecting tests to run

Running the tool without any argument runs only the *default tests*.

Those include all tests that are known to be safe to be run concurrently as part of a single run.

Tests that either change the global state of the ledger (e.g. configuration management) or are designed to stress the implementation need to be explicitly included using the available command line options.

Use the following command line flags to select which tests to run:

> `--list`: print all available test suites to the console, shows if they are run by default
> `--list-all`: print all available tests to the console, shows if they are run by default
> `--include`: only run the tests that match the argument
> `--exclude`: do not run the tests that match the argument
> `--perf-tests`: list performance tests to run; cannot be combined with normal tests

Include and exclude are matched as prefixes, e.g. `--exclude=SemanticTests` will exclude all tests whose name starts with `SemanticTests`. Test names always start with their suite name followed by a colon, so the test suite names shown by `--list` can be useful for coarse-grained inclusion/exclusion.

Both `--include` and `--exclude` (and `--perf-tests`) can be specified multiple times and/or provide comma-separated lists, i.e. all of these are equivalent:

```
--include=a,b,c
--include=a --include=b --include=c
--include=a,b --include=c
```

The logic is always to first select included tests, then remove from that the excluded ones, i.e. include directives never override a corresponding exclude directive.

If no `--include` flag is given, all of the tests are included. You cannot run performance and non-performance tests in the same invocation. `--exclude` is ignored when running performance tests, and the program will stop if it detects that both `--perf-tests` and `--include` have been specified.

Examples (hitting a single participant at `localhost:6865`):

Listing 1: Only run `TestA`

```
$ java -jar ledger-api-test-tool.jar --include TestA localhost:6865
```

Listing 2: Run all tests, but not `TestB`

```
$ java -jar ledger-api-test-tool.jar --exclude TestB localhost:6865
```

Listing 3: Run all tests

```
$ java -jar ledger-api-test-tool.jar localhost:6865
```

Listing 4: Run all tests, but not `TestC`

```
$ java -jar ledger-api-test-tool.jar --exclude TestC
```

Performance tests

The available performance tests allow to establish the performance envelope of the ledger under test (a term borrowed from aeronautics), which offers an indication of the amount of the parameters under which a ledger implementation is supposed to perform.

Those tests include tail latency, throughput and maximum size of a single transaction. You can run the tool with the `--list` option to see a list of available test suites that includes individual performance envelope test cases. You can mix and match those tests to produce a test suite tailored to match the expected performance envelope of a given ledger implementation using a specific hardware setup.

For example, the following will verify that the ledger under test can have a tail latency of one second when processing twenty pings, perform twenty pings per seconds and being able to process a transaction one megabyte in size:

```
$ java -jar ledger-api-test-tool.jar \
  --perf-tests=PerformanceEnvelope.Latency.1000ms \
  --perf-tests=PerformanceEnvelope.Throughput.TwentyOPS \
  --perf-tests=PerformanceEnvelope.TransactionSize.1000KB \
  localhost:6865
```

Note: A ping is a collective name for two templates used to evaluate the performance envelope. Each of the two templates, Ping and Pong , have a single choice allowing the controller to create an instance of the complementary template, directed to the original sender.

The test run will also produce a short summary of statistics which is printed to standard output by default but that can be written to a specific file path using the `--perf-tests-report` command line option.

8.3.1.4 Try out the Ledger API Test Tool against DAML Sandbox

If you wanted to test out the tool, you can run it against *DAML Sandbox*. To do this:

```
$ java -jar ledger-api-test-tool.jar --extract
$ daml sandbox *.dar
$ java -jar ledger-api-test-tool.jar localhost:6865
```

This should always succeed, as the Sandbox is tested to correctly implement the Ledger API. This is useful if you do not have yet a custom Ledger API endpoint.

8.3.1.5 Using the tool with a known-to-be-faulty Ledger API implementation

Use flag `--must-fail` if you expect one or more or the scenario tests to fail. If enabled, the tool will return the success exit code when at least one test fails, and it will return a failure exit code when all tests succeed:

```
java -jar ledger-api-test-tool.jar --must-fail localhost:6865
```

This is useful during development of a DAML ledger implementation, when tool needs to be used against a known-to-be-faulty implementation (e.g. in CI). It will still print information about failed tests.

8.3.1.6 Tuning the testing behaviour of the tool

Use the command line option `--timeout-scale-factor` to tune timeouts applied by the tool.

Set `--timeout-scale-factor` to a floating point value higher than 1.0 to make the tool wait longer for expected events coming from the DAML ledger implementation under test. Conversely use values smaller than 1.0 to make it wait shorter.

8.3.1.7 Accomodating different ledger clock intervals

Use the command line option `--ledger-clock-granularity` to indicate the maximum interval at which the ledger's clock will increment.

If running on a ledger where ledger time increments in a time period greater than 10s, set `--ledger-clock-granularity` to a value higher than 10000 (10,000ms). Tests that are sensitive to the ledger clock will then wait for a corresponding longer period of time to ensure completion of operations, avoiding timeouts and premature failures. The command deduplication test suite is particularly sensitive to this value.

8.3.1.8 Verbose output

Use the command line option `--verbose` to print full stack traces on failures.

8.3.1.9 Concurrent test runs

To minimize concurrent runs of tests, `--concurrent-test-runs` can be set to 1 or 2. The default value is the number of processors available.

Note that certain tests, known to be possibly interfering with others (e.g. configuration management), are always run sequentially and as the last tests in a run.

8.3.1.10 Retired tests

A few tests can be retired over time as they could be deemed not providing the necessary signal to a developer or operator that an integration correctly implements the DAML Ledger API. Those test will nominally be kept in the test suite for a time to prevent unwanted breakages of existing CI pipelines. They will however not be run and they will eventually be removed. You are advised to remove any explicit reference to those tests while they are in their deprecation period.

Retired tests are not listed when using `--list` or `--list-all` but can be included in a run using `--include`. In this case, nothing will be run and the test report will mention that the test has been retired and skipped.

DAML Applications run on DAML Ledgers. A DAML Ledger is a server serving the *Ledger API* as per the semantics defined in the *DAML Ledger Model* and the DAML-LF specification.

The DAML Integration Kit helps third-party ledger developers to implement a DAML Ledger on top of their distributed ledger or database of choice.

We provide the resources in the kit, which include guides to

DAML Integration Kit status and roadmap

Implementing your own DAML Ledger
Deploying a DAML Ledger
Testing a DAML Ledger
Benchmarking a DAML Ledger

Using these guides, you can focus on your own distributed-ledger or database and reuse our DAML Ledger server and DAML interpreter code for implementing the DAML Ledger API. For example uses of the integration kit, see below.

8.3.2 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

8.3.3 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*.

8.3.3.1 Step-by-step guide

Prerequisite knowledge

Before you can decide on an appropriate architecture and implement your own server and validator, you need a significant amount of context about DAML. To acquire this context, you should:

1. Complete the *IOU Quickstart Tutorial*.
2. Get an in-depth understanding of the *DAML Ledger Model*.
3. Build a mental model of how the *Ledger API* is used to *build DAML Applications*.

Deciding on the architecture and writing the code

Once you have the necessary context, we recommend the steps to implement your own server and validator:

1. Clone our example DAML Ledger (which is backed by an in-memory key-value store) from the digital-asset/daml-on-x-example.

1. Read the example code jointly with the *Architectural overview*, *Resources we provide*, and the *Library infrastructure overview* below.

1. Combine all the knowledge gained to decide on the architecture for your DAML on *X* ledger.

1. Implement your architecture; and let the world know about it by creating a PR against the digital-asset/daml repository to add your ledger to the list of *DAML Ledgers built or in development*.

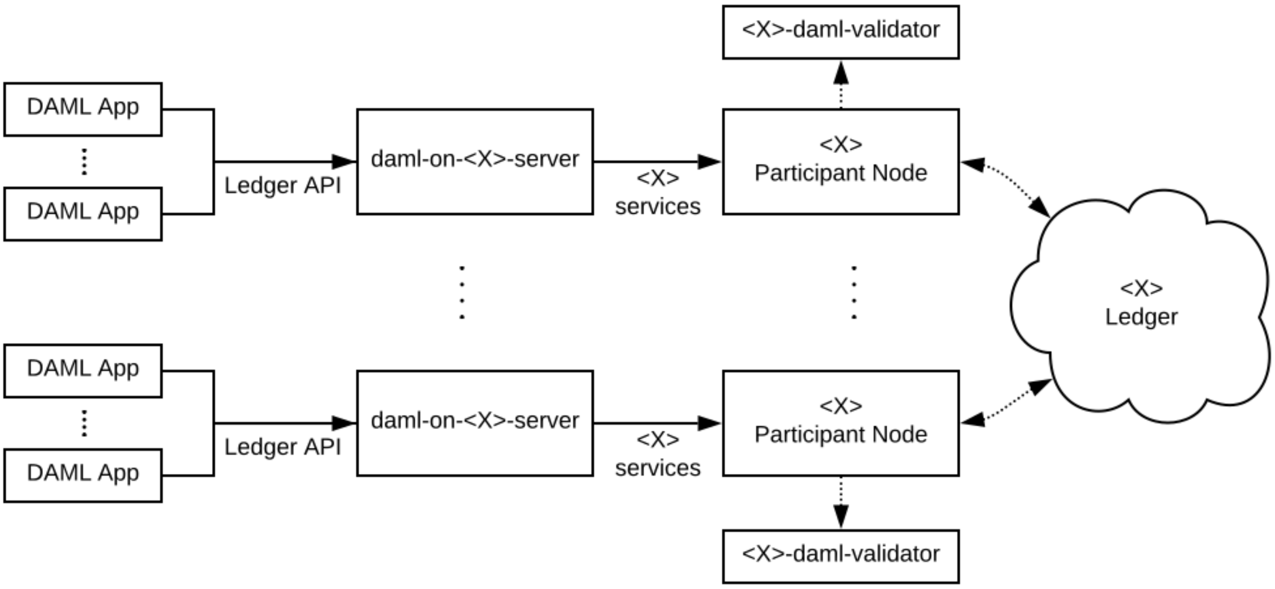
If you need help, then feel free to use the feedback form on this documentation page or GitHub issues on the digital-asset/daml repository to get into contact with us.

8.3.3.2 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)

8.3.3.3 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.

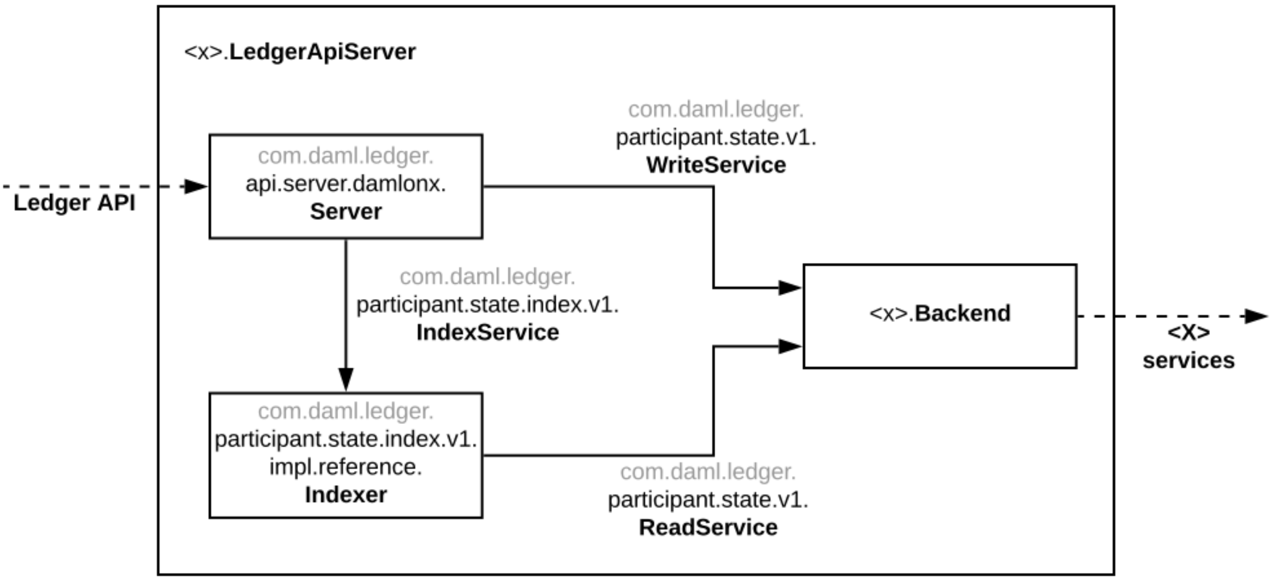
`participant-state-kvutils.jar` (source code) These utilities provide methods to succintly implement interfaces from `participant-state.jar` on top of a key-value state storage.
See documentation in package.scala

`ledger-api-server.jar` (source code for API server, source code for indexer) Contains code that implements a DAML Ledger API server and the SQL-backed indexer given implementations of the interfaces in `participant-state.jar`.

`daml-engine.jar` (source code) Contains code for serializing and deserializing DAML transactions and for validating them.
An *<X>-daml-validator* is typically implemented by wrapping this code in the X-ledger's SDK for building transaction validators. `daml-engine.jar` also contains code for interpreting commands sent over the Ledger API. It is used by the *daml-on-<X>-server* to construct the transactions submitted to its participant node.

This diagram shows how the classes and interfaces provided by these libraries are typically combined to instantiate a DAML Ledger API server backed by an X ledger:

TODO: Update this diagram to mention ledger server classes above instead of deprecated daml-on-x-server



In the diagram above:

Boxes labeled with fully qualified class names denote class instances.
Solid arrows labeled with fully qualified interface names denote that an instance depends on another instance providing that interface.
Dashed arrows denote that a class instance provides or depends on particular services.

Boxes embedded in other boxes denote that the outer class instance creates the contained instances.

Explaining this diagram in detail (for brevity, we drop prefixes of their qualified names where unambiguous):

Ledger API is the collection of gRPC services that you would like your *daml-on-<X>-server* to provide.

<X> services are the services provided by which underly your ledger, which you aim to leverage to build your *daml-on-<X>-server*.

<x>.LedgerApiServer is the class whose main method or constructor creates the contained instances and wires them up to provide the Ledger API backed by the `<X> services`. You need to implement this for your DAML on X ledger.

WriteService (source code) is an interface abstracting over the mechanism to submit DAML transactions to the underlying X ledger via a participant node.

ReadService (source code) is an interface abstracting over the ability to subscribe to changes of the X ledger visible to a particular participant node. The changes are exposed as a stream that is resumable from any particular offset, which supports restarts of the consumer. We typically expect there to be a single consumer of the data provided on this interface. That consumer is responsible for assembling the streamed changes into a view onto the participant state suitable for querying.

<x>.Backend is a class implementing the `ReadService` and the `WriteService` on top of the `<X> services`. You need to implement this for your DAML on X ledger.

StandaloneIndexerServer (source code) is a standalone service that subscribe to ledger changes using `ReadService` and inserts the data into a SQL backend (index) for the purpose of serving the data over the Ledger API.

StandaloneIndexServer (source code) is a class containing all the code to implement the Ledger API on top of an ledger backend. It serves the data from a SQL database populated by the `StandaloneIndexerServer`.

8.3.4 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).

8.3.4.1 Authorization

To implement authorization on your ledger, do the following modifications to your code:

Implement the `com.daml.ledger.api.auth.AuthService` (source code) interface. An AuthService receives all HTTP headers attached to a gRPC ledger API request and returns a set of `Claims` (source code), which describe the authorization of the request.
Instantiate a `com.daml.ledger.api.auth.interceptor.AuthorizationInterceptor` (source code), and pass it an instance of your AuthService implementation. This interceptor will be responsible for storing the decoded Claims in a place where ledger API services can access them.
When starting the `com.daml.platform.apiserver.LedgerApiServer` (source code), add the above AuthorizationInterceptor to the list of interceptors (see `interceptors` parameter of `LedgerApiServer.create`).

For reference, you can have a look at how authorization is implemented in the sandbox:

> The `com.daml.ledger.api.auth.AuthServiceJWT` class (source code) reads a JWT token from HTTP headers.
> The `com.daml.ledger.api.auth.AuthServiceJWTPayload` class (source code) defines the format of the token payload.
> The token signature algorithm and the corresponding public key is specified as a sandbox command line parameter.

8.3.5 Testing a DAML Ledger

You can test your DAML ledger implementation using *Ledger API Test Tool*, which will assess correctness of implementation of the *Ledger API*. For example, it will show you if there are consistency or conformance problem with your implementation.

Assuming that your Ledger API endpoint is accessible at `localhost:6865`, you can use the tool in the following manner:

1. Download the Ledger API Test Tool from Maven and save it as `ledger-api-test-tool.jar` in your current directory.
2. Obtain the DAML archives required to run the tests:
   `java -jar ledger-api-test-tool.jar --extract`
3. Load all `.dar` files extracted in the current directory into your Ledger.
4. Run the tool against your ledger:
   `java -jar ledger-api-test-tool.jar localhost:6865`

See more in *Ledger API Test Tool*.

8.3.6 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).

8.4 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 *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* -> Time -> Map *CommandId* [*Command*] -> s -> *TriggerA* ()	The rule defines the main logic of your trigger. It can send commands to the ledger using `emitCommands` to change the ACS. The rule depends on the following arguments: * The party your trigger is running as. * The current state of the ACS. * The current time (UTC in wallclock mode, Unix epoch in static mode) * The commands in flight. * The user-defined state.
registeredTemplates	*RegisteredTemplates*	The templates the trigger will receive events for.
heartbeat	Optional RelTime	Send a heartbeat message at the given interval.

instance HasField heartbeat (*Trigger* s) (Optional RelTime)

instance HasField initialize (*Trigger* s) (*ACS* -> s)

instance HasField registeredTemplates (*Trigger* s) *RegisteredTemplates*

instance HasField rule (*Trigger* s) (Party -> *ACS* -> Time -> Map *CommandId* [*Command*] -> s -> *TriggerA* ())

instance HasField updateState (*Trigger* s) (*ACS* -> *Message* -> s -> s)

Functions

getTemplates : Template a => *ACS* -> [(ContractId a, a)]

getContracts : Template a => *ACS* -> [(ContractId a, a)]
Extract the contracts of a given template from the ACS.

emitCommands : [*Command*] -> [*AnyContractId*] -> *TriggerA CommandId*
Send a transaction consisting of the given commands to the ledger. The second argument can be used to mark a list of contract ids as pending. These contracts will automatically be filtered from getContracts until we either get the corresponding transaction event for this command or a failing completion.

dedupCreate : (Eq t, Template t) => t -> *TriggerA* ()
Create the template if it's not already in the list of commands in flight (it will still be created if it is in the ACS).

Note that this will send the create as a single-command transaction. If you need to send multiple commands in one transaction, use `emitCommands` with `createCmd` and handle filtering yourself.

dedupCreateAndExercise : (Eq t, Eq c, Template t, Choice t c r) => t -> c -> *TriggerA* ()

Create the template and exercise a choice on it it's not already in the list of commands in flight (it will still be created if it is in the ACS).

Note that this will send the create and exercise as a single-command transaction. If you need to send multiple commands in one transaction, use `emitCommands` with `createAndExerciseCmd` and handle filtering yourself.

dedupExercise : (Eq c, Choice t c r) => ContractId t -> c -> *TriggerA* ()

Exercise the choice on the given contract if it is not already in flight.

Note that this will send the exercise as a single-command transaction. If you need to send multiple commands in one transaction, use `emitCommands` with `exerciseCmd` and handle filtering yourself.

If you are calling a consuming choice, you might be better off by using `emitCommands` and adding the contract id to the pending set.

dedupExerciseByKey : (Eq c, Eq k, Choice t c r, TemplateKey t k) => k -> c -> *TriggerA* ()

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.Assert

Data Types

data *ACSBuilder*

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

instance Monoid *ACSBuilder*

instance Semigroup *ACSBuilder*

Functions

toACS : Template t => ContractId t -> *ACSBuilder*

Include the given contract in the 'ACS'.

testRule : *Trigger* s -> Party -> *ACSBuilder* -> Map *CommandId* [*Command*] -> s -> Script [*Commands*]

Execute a trigger's rule once in a scenario.

flattenCommands : [*Commands*] -> [*Command*]

Drop 'CommandId's and extract all 'Command's.

assertCreateCmd : (Template t, CanAbort m) => [*Command*] -> (t -> Either Text ()) -> m ()

Check that at least one command is a create command whose payload fulfills the given assertions.

assertExerciseCmd : (Template t, Choice t c r, CanAbort m) => [*Command*] -> ((ContractId t, c) -> Either Text ()) -> m ()

Check that at least one command is an exercise command whose contract id and choice argument fulfill the given assertions.

assertExerciseByKeyCmd : (TemplateKey t k, Choice t c r, CanAbort m) => [*Command*] -> ((k, c) -> Either Text ()) -> m ()

Check that at least one command is an exercise by key command whose key and choice argument fulfill the given assertions.

8.4.1.3 Module Daml.Trigger.Internal

Data Types

data *ACS*

Active contract set, you can use `getContracts` to access the templates of a given type.

ACS

Field	Type	Description
activeContracts	[(*AnyContractId*, AnyTemplate)]	
pendingContracts	Map *CommandId* [*AnyContractId*]	

instance HasField acs (*TriggerState* s) *ACS*

instance HasField activeContracts *ACS* [(*AnyContractId*, AnyTemplate)]

instance HasField initialize (*Trigger* s) (*ACS* -> s)

instance HasField pendingContracts *ACS* (Map *CommandId* [*AnyContractId*])

instance HasField rule (*Trigger* s) (Party -> *ACS* -> Time -> 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.

TriggerA (State *TriggerAState* a)

instance Functor *TriggerA*

instance Action *TriggerA*

instance Applicative *TriggerA*

instance HasField rule (*Trigger* s) (Party -> *ACS* -> Time -> Map *CommandId* [*Command*] -> s -> *TriggerA* ())

data *TriggerAState*

TriggerAState

Field	Type	Description
commandsInFlight	Map CommandId [Command]	
emittedCommands	[Commands]	Emitted commands in reverse because I can't be bothered to implement a dlist.
pendingContracts	Map CommandId [AnyContractId]	Map from command ids to the contract ids marked pending by that command.
nextCommandId	Int	Command id used for the next submit

instance HasField commandsInFlight *TriggerAState* (Map *CommandId* [*Command*])

instance HasField emittedCommands *TriggerAState* [*Commands*]

instance HasField nextCommandId *TriggerAState* Int

instance HasField pendingContracts *TriggerAState* (Map *CommandId* [*AnyContractId*])

data *TriggerState* s

TriggerState

Field	Type	Description
acs	ACS	
party	Party	
userState	s	
commandsInFlight	Map CommandId [Command]	
nextCommandId	Int	

instance HasField acs (*TriggerState* s) *ACS*

instance HasField commandsInFlight (*TriggerState* s) (Map *CommandId* [*Command*])

instance HasField nextCommandId (*TriggerState* s) Int

instance HasField party (*TriggerState* s) Party

instance HasField userState (*TriggerState* s) s

Functions

addCommands : Map *CommandId* [*Command*] -> *Commands* -> Map *CommandId* [*Command*]

insertTpl : *AnyContractId* -> AnyTemplate -> *ACS* -> *ACS*

deleteTpl : *AnyContractId* -> *ACS* -> *ACS*

lookupTpl : Template a => *AnyContractId* -> *ACS* -> Optional a

applyEvent : *Event* -> *ACS* -> *ACS*

applyTransaction : *Transaction* -> *ACS* -> *ACS*

runRule : (Party -> *ACS* -> Time -> Map *CommandId* [*Command*] -> s -> *TriggerA* ()) -> Time -> *TriggerState* s -> (*TriggerState* s, [*Commands*])

runTriggerA : *TriggerA* a -> *TriggerAState* -> (a, *TriggerAState*)

8.4.1.4 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 -> Time -> *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 activeContracts *ACS* [(*AnyContractId*, AnyTemplate)]

instance HasField contractId *AnyContractId* (ContractId ())

instance HasField contractId *Archived AnyContractId*

instance HasField contractId *Command AnyContractId*

instance HasField contractId *Created AnyContractId*

instance HasField pendingContracts *ACS* (Map *CommandId* [*AnyContractId*])

instance HasField pendingContracts *TriggerAState* (Map *CommandId* [*AnyContractId*])

instance HasField templateId *AnyContractId* TemplateTypeRep

data *Archived*

The data in an `Archived` event.

Archived

Field	Type	Description
eventId	*EventId*	
contractId	*AnyContractId*	

instance Eq *Archived*

instance Show *Archived*

instance HasField contractId *Archived AnyContractId*

instance HasField eventId *Archived EventId*

data *Command*

A ledger API command. To construct a command use `createCmd` and `exerciseCmd`.

CreateCommand

Field	Type	Description
templateArg	AnyTemplate	

ExerciseCommand

Field	Type	Description
contractId	*AnyContractId*	
choiceArg	AnyChoice	

CreateAndExerciseCommand

Field	Type	Description
templateArg	AnyTemplate	
choiceArg	AnyChoice	

ExerciseByKeyCommand

Field	Type	Description
tplTypeRep	TemplateTypeRep	
contractKey	AnyContractKey	
choiceArg	AnyChoice	

instance HasField choiceArg *Command* AnyChoice

instance HasField commands *Commands* [*Command*]

instance HasField commandsInFlight *TriggerAState* (Map *CommandId* [*Command*])

instance HasField commandsInFlight (*TriggerState* s) (Map *CommandId* [*Command*])

instance HasField contractId *Command AnyContractId*

instance HasField "contractKey" *Command* AnyContractKey

instance HasField "rule" (*Trigger* s) (Party -> *ACS* -> Time -> Map *CommandId* [*Command*] -> s -> *TriggerA* ())

instance HasField "templateArg" *Command* AnyTemplate

instance HasField "tplTypeRep" *Command* TemplateTypeRep

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 "pendingContracts" *ACS* (Map *CommandId* [*AnyContractId*])

instance HasField "pendingContracts" *TriggerAState* (Map *CommandId* [*AnyContractId*])

instance HasField "rule" (*Trigger* s) (Party -> *ACS* -> Time -> 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 -> Time -> *ActiveContracts* -> (s, [*Commands*]))

instance HasField "update" (*Trigger* s) (Time -> *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	*CompletionStatus*	

instance Show *Completion*

instance HasField commandId *Completion CommandId*

instance HasField status *Completion CompletionStatus*

data *CompletionStatus*

Failed

Field	Type	Description
status	Int	
message	Text	

Succeeded

Field	Type	Description
transactionId	*TransactionId*	

instance Show *CompletionStatus*

instance HasField message *CompletionStatus* Text

instance HasField status *Completion CompletionStatus*

instance HasField status *CompletionStatus* Int

instance HasField transactionId *CompletionStatus TransactionId*

data *Created*

The data in a `Created` event.

Created

Field	Type	Description
eventId	*EventId*	
contractId	*AnyContractId*	
argument	AnyTemplate	

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

instance HasField "argument" *Created* AnyTemplate

instance HasField "contractId" *Created* *AnyContractId*

instance HasField "eventId" *Created* *EventId*

data *Event*

An event in a transaction. This definition should be kept consistent with the object `EventVariant` defined in triggers/runner/src/main/scala/com/digitalasset/daml/lf/engine/trigger/Converter.scala

CreatedEvent *Created*

ArchivedEvent *Archived*

instance HasField "events" *Transaction* [*Event*]

data *EventId*

EventId Text

instance Eq *EventId*

instance Show *EventId*

instance HasField "eventId" *Archived* *EventId*

instance HasField "eventId" *Created* *EventId*

data *Message*

Either a transaction or a completion. This definition should be kept consistent with the object `MessageVariant` defined in triggers/runner/src/main/scala/com/digitalasset/daml/lf/engine/trigger/Converter.scala

MTransaction *Transaction*

MCompletion *Completion*

MHeartbeat

instance HasField "update" (*Trigger* s) (Time -> *Message* -> s -> (s, [*Commands*]))

instance HasField "updateState" (*Trigger* s) (*ACS* -> *Message* -> s -> s)

data *RegisteredTemplates*

AllInDar

Listen to events for all templates in the given DAR.

RegisteredTemplates [RegisteredTemplate]

instance HasField "registeredTemplates" (*Trigger* s) *RegisteredTemplates*

instance HasField "registeredTemplates" (*Trigger* s) *RegisteredTemplates*

data *Transaction*

Transaction

Field	Type	Description
transactionId	TransactionId	
commandId	Optional CommandId	
events	[Event]	

instance HasField commandId *Transaction* (Optional *CommandId*)

instance HasField events *Transaction* [*Event*]

instance HasField transactionId *Transaction TransactionId*

data *TransactionId*

TransactionId Text

instance Eq *TransactionId*

instance Show *TransactionId*

instance HasField transactionId *CompletionStatus TransactionId*

instance HasField transactionId *Transaction TransactionId*

data *Trigger* s

Trigger is (approximately) a left-fold over `Message` with an accumulator of type `s`.

Trigger

Field	Type	Description
initialState	Party -> Time -> ActiveContracts -> (s, [Commands])	
update	Time -> Message -> s -> (s, [Commands])	
registeredTemplates	RegisteredTemplates	
heartbeat	Optional RelTime	

instance HasField heartbeat (*Trigger* s) (Optional RelTime)

instance HasField initialState (*Trigger* s) (Party -> Time -> *ActiveContracts* -> (s, [*Commands*]))

instance HasField registeredTemplates (*Trigger* s) *RegisteredTemplates*

instance HasField update (*Trigger* s) (Time -> *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.

registeredTemplate : Template t => RegisteredTemplate

createCmd : Template t => t -> *Command*
Create a contract of the given template.

exerciseCmd : Choice t c r => ContractId t -> c -> *Command*
Exercise the given choice.

createAndExerciseCmd : (Template t, Choice t c r) => t -> c -> *Command*
Create a contract of the given template and immediately exercise the given choice on it.

exerciseByKeyCmd : (Choice t c r, TemplateKey t k) => k -> c -> *Command*

fromCreate : Template t => *Command* -> Optional t
Check if the command corresponds to a create command for the given template.

fromCreateAndExercise : (Template t, Choice t c r) => *Command* -> Optional (t, c)
Check if the command corresponds to a create and exercise command for the given template.

fromExercise : Choice t c r => *Command* -> Optional (ContractId t, c)
Check if the command corresponds to an exercise command for the given template.

fromExerciseByKey : (Choice t c r, TemplateKey t k) => *Command* -> Optional (k, c)
Check if the command corresponds to an exercise by key command for the given template.

WARNING: DAML Triggers are an early access feature that is actively being designed and is *subject to breaking changes*. We welcome feedback about DAML triggers on our issue tracker, our forum, 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
    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 -> Time -> Map CommandId [Command] -> s ->□
↪TriggerA ()
  , registeredTemplates : RegisteredTemplates
  , heartbeat : Optional RelTime
  }
```

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.

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 current time, 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`.

We can specify the templates that our trigger will operate on. In our case, we will simply specify `AllInDar` which means that the trigger will receive events for all template types defined in the DAR. It is also possible to specify an explicit list of templates, e.g., `RegisteredTemplates [registeredTemplate @Original, registeredTemplate @Subscriber, registeredTemplate @Copy]`. This is mainly useful for performance reasons if your DAR contains many templates that are not relevant for your trigger.

Finally, you can specify an optional heartbeat interval at which the trigger will be sent a `MHeartbeat` message. This is useful if you want to ensure that the trigger is executed at a certain rate to issue timed commands.

For our DAML trigger, the definition looks as follows:

```
copyTrigger : Trigger ()
copyTrigger = Trigger
  { initialize = \_acs -> ()
  , updateState = \_acs _message () -> ()
  , rule = copyRule
  , registeredTemplates = AllInDar
  , heartbeat = None
  }
```

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 -> Time -> Map CommandId [Command] -> () ->□
↪TriggerA ()
copyRule party acs _time 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 getContracts function that given the ACS will return a list of all contracts of a given template.

```
  let subscribers : [(ContractId Subscriber, Subscriber)] = getContracts□
↪@Subscriber acs
  let originals : [(ContractId Original, Original)] = getContracts□
↪@Original acs
  let copies : [(ContractId Copy, Copy)] = getContracts @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 archive 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`. In addition to that, we also pass in a list of contract ids. Those contracts will be marked pending and not be included in the result of `getContracts` until the commands have either been comitted to the ledger or the command submission failed.

```
  forA archiveCopies $ \cid -> emitCommands [exerciseCmd cid Archive]
↪[toAnyContractId cid]
```

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 --template 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, the party that our trigger is executed as, and the time mode of the ledger which is the sandbox default, i.e, static time.

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.

When using DAML triggers against a Ledger with authentication, you can pass `--access-token-file token.jwt` to `daml trigger` which will read the token from the file `token.jwt`.

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.

8.5 Visualizing DAML Contracts

You can generate visual graphs for the contracts in your DAML project. To do this:

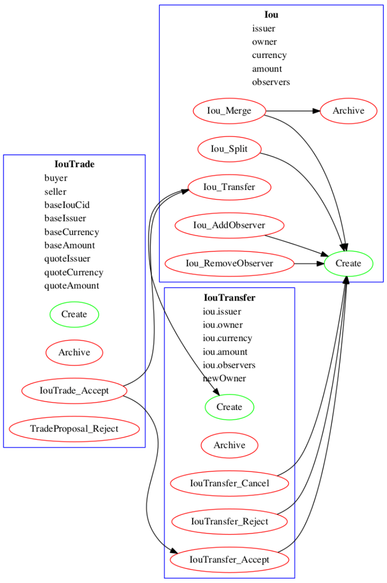
1. Install Graphviz.
2. Generate a DAR from your project by running `daml build`.
3. Generate a dot file from that DAR by running `daml damlc visual <path_to_project>/dist/<project_name.dar> --dot <project_name>.dot`
4. Generate the visual graph with Graphviz by running `dot -Tpng <project_name>.dot > <project_name>.png`

8.5.1 Example: Visualizing the Quickstart project

Here's an example visualization based on the *quickstart*. You'll need to install Graphviz to try this out.

1. Generate the dar using `daml build`
2. Generate a dot file `daml damlc visual dist/quickstart-0.0.1.dar --dot quickstart.dot`
3. Generate the visual graph with Graphviz by running `dot -Tpng quickstart.dot -o quickstart.png`

Running the above should produce an image which looks something like this:



8.5.2 Visualizing DAML Contracts - Within IDE

You can generate visual graphs from VS Code IDE. Open the daml project in VS Code and use command palette. Should reveal a new window pane with dot image. Also visual generates only the currently open daml file and its imports.

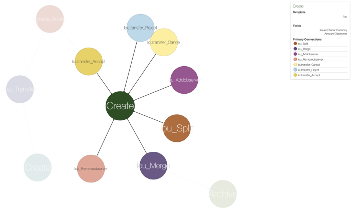
Note: You will need to install the Graphviz/dot packages as mentioned above.

8.5.3 Visualizing DAML Contracts - Interactive Graphs

This does not require any packages installed. You can generate D3 graphs for the contracts in your DAML project. To do this

1. Generate a DAR from your project by running `daml build`
2. Generate HTML file `daml damlc visual-web .daml/dist/quickstart-0.0.1.dar -o quickstart.html`

Running the above should produce an image which looks something like this:



Chapter 9

Support and updates

9.1 Support

Have questions or feedback? You're in the right place.

> **Questions: Forum**
> For how do I? , why does something work this way or I've got a programming problem I'm trying to solve questions, the `Questions` category on our forum is the best place to ask.
> If you're not sure what makes a good question, take a look at our guide on the topic.
>
> **Feedback: Forum**
> If you want to give feedback, you can make a topic in the `General` category on our forum or on Slack in the `#public` channel.

When you're in the community Forum or on Stack Overflow, please keep to our Code of Conduct.

9.1.1 Support expectations

For community users (ie on our Forum and Stack Overflow):

> **Timing**: You can enjoy the support of the community, which is provided for you out of their own good will and free time. On top of that, a Digital Asset employee will try to reply to unanswered questions within two business days.
> Business days are affected by public holidays. Engineers contributing to DAML are mostly located in Zurich and New York, so please be mindful of the public holidays in those locations (timeanddate.com maintains an unofficial list of holidays for both Switzerland and the United States).
>
> **Public support**: We only offer public support - for example, in the `Questions` category on our forum or on Slack in the `#public` channel.
> We can't answer questions in private messages or over email, so please only ask questions in public forums.
>
> **Level of support**: We're happy to answer questions about error messages you're encountering, or discuss DAML design questions. However, we can't provide more extensive consultation on how to build your DAML application or the languages, frameworks, libraries and tools you may use to build it.

If you need private support, or want consultation from Digital Asset about how to build your DAML application, they offer paid support. Please contact Digital Asset to ask about pricing.

9.2 Release notes

Release notes are now hosted on the DAML blog.

9.3 DAML roadmap (as of July 2020)

A new roadmap is currently being worked on and will be published soon. If you have requests for features, see the *Support* page for how to get in touch.