

# DAML SDK Documentation

**DAML**

**Digital Asset**

Version : 2019-04-01

# Table of contents

<b>Table of contents</b>	i
<b>1 Getting Started</b>	1
1.1 Installing the SDK . . . . .	1
1.1.1 1. Install the dependencies . . . . .	1
1.1.2 2. Set up the SDK Assistant . . . . .	1
1.1.3 3. Configure Maven . . . . .	1
1.1.4 Next steps . . . . .	2
1.2 Introduction to the product . . . . .	2
1.2.1 Distributed business processes . . . . .	2
1.2.2 DAML . . . . .	4
1.2.3 Digital Asset Ledger Server . . . . .	6
1.2.4 DAML SDK . . . . .	7
1.2.5 Summary and next steps . . . . .	7
1.3 Quickstart guide . . . . .	7
1.3.1 Download the quickstart application . . . . .	8
1.3.2 Overview of what an IOU is . . . . .	9
1.3.3 Run the application using prototyping tools . . . . .	10
1.3.4 Try out the application . . . . .	11
1.3.5 Get started with DAML . . . . .	15
1.3.6 Integrate with the ledger . . . . .	20
1.3.7 Next steps . . . . .	23
<b>2 Modeling Processes with DAML</b>	24
2.1 DAML reference docs . . . . .	24
2.1.1 Overview: template structure . . . . .	24
2.1.2 Reference: templates . . . . .	26
2.1.3 Reference: choices . . . . .	28
2.1.4 Reference: updates . . . . .	30
2.1.5 Reference: data types . . . . .	34
2.1.6 Reference: built-in functions . . . . .	40
2.1.7 Reference: expressions . . . . .	43
2.1.8 Reference: functions . . . . .	45
2.1.9 Reference: scenarios . . . . .	48
2.1.10 Reference: DAML file structure . . . . .	49
2.2 DAML Standard Library . . . . .	51
2.2.1 Usage . . . . .	51
2.2.2 Domains . . . . .	51
2.3 DAML Studio . . . . .	86
2.3.1 Creating your first DAML file using DAML Studio . . . . .	86

2.3.2	Supported features . . . . .	88
2.3.3	Common scenario errors . . . . .	90
2.4	Testing DAML using scenarios . . . . .	93
2.4.1	Scenario syntax . . . . .	94
2.4.2	Running scenarios in DAML Studio . . . . .	95
2.4.3	Examples . . . . .	95
2.5	Troubleshooting DAML . . . . .	97
2.5.1	Error: <X> is not authorized to commit an update . . . . .	97
2.5.2	Error Argument is not of serializable type . . . . .	97
2.5.3	Modelling questions . . . . .	98
2.5.4	Testing questions . . . . .	100
2.6	Converting a project to DAML 1.2 . . . . .	100
2.6.1	Introduction: the upgrade process . . . . .	101
2.6.2	Start converting . . . . .	102
2.6.3	Templates . . . . .	102
2.6.4	Choices . . . . .	103
2.6.5	Scenarios . . . . .	106
2.6.6	Built-in types and functions . . . . .	107
2.6.7	Syntax . . . . .	110
2.6.8	Other changes . . . . .	111
2.6.9	Other errors . . . . .	113
2.7	Writing good DAML . . . . .	113
2.7.1	DAML design patterns . . . . .	113
2.7.2	DAML anti-patterns . . . . .	131
2.7.3	What functionality belongs in DAML models versus application code? . . . . .	135
<b>3</b>	<b>Building Applications</b> . . . . .	<b>137</b>
3.1	Building applications against a DA ledger . . . . .	137
3.1.1	Categories of application . . . . .	137
3.1.2	The Ledger API . . . . .	138
3.1.3	Structuring an application . . . . .	141
3.1.4	Application Libraries . . . . .	143
3.1.5	Architecture Guidance . . . . .	144
3.2	Ledger API . . . . .	146
3.2.1	Ledger API Reference . . . . .	146
3.2.2	From DAML to Ledger API . . . . .	178
3.2.3	gRPC . . . . .	185
3.2.4	Getting started . . . . .	185
3.2.5	Service layers . . . . .	186
3.2.6	Transaction and transaction trees . . . . .	187
3.2.7	DAML-LF . . . . .	187
3.2.8	Commonly used types . . . . .	188
3.2.9	Error handling . . . . .	188
3.2.10	Verbosity . . . . .	188
3.2.11	Limitations . . . . .	189
3.2.12	Versioning . . . . .	189
3.2.13	Authentication . . . . .	189
3.3	Java Binding . . . . .	189
3.3.1	Generating Java code from DAML . . . . .	189
3.3.2	Overview . . . . .	197
3.3.3	Getting started . . . . .	198

3.3.4	The underlying library: RxJava . . . . .	201
3.4	How are DAML types translated to DAML-LF? . . . . .	201
3.4.1	What is DAML-LF? . . . . .	202
3.4.2	When you need to know about DAML-LF . . . . .	202
3.4.3	Translation of DAML types to DAML-LF . . . . .	202
<b>4</b>	<b>SDK Tools</b>	<b>207</b>
4.1	SDK Assistant . . . . .	207
4.1.1	Installing the SDK Assistant . . . . .	207
4.1.2	Understanding the SDK Assistant . . . . .	207
4.1.3	Using the SDK Assistant . . . . .	208
4.1.4	Developing with the SDK Assistant . . . . .	209
4.1.5	Building DAML archives . . . . .	210
4.1.6	Managing SDK releases . . . . .	210
4.1.7	Managing SDK templates . . . . .	211
4.1.8	Running SDK tools . . . . .	212
4.1.9	da.yaml configuration . . . . .	212
4.1.10	Uninstalling DAML SDK . . . . .	213
4.2	DAML Sandbox . . . . .	213
4.2.1	Command-line reference . . . . .	214
4.2.2	Performance benchmarks . . . . .	215
4.3	Navigator . . . . .	215
4.3.1	Navigator functionality . . . . .	215
4.3.2	Installing and starting Navigator . . . . .	215
4.3.3	Choosing a party / changing the party . . . . .	216
4.3.4	Logging out . . . . .	217
4.3.5	Viewing templates or contracts . . . . .	217
4.3.6	Using Navigator . . . . .	220
4.3.7	Advanced usage . . . . .	223
<b>5</b>	<b>Background Concepts</b>	<b>226</b>
5.1	DA Ledger Model . . . . .	226
5.1.1	Structure . . . . .	227
5.1.2	Integrity . . . . .	231
5.1.3	Privacy . . . . .	243
5.1.4	DAML: Defining Contract Models Compactly . . . . .	251
<b>6</b>	<b>Examples</b>	<b>253</b>
6.1	DAML examples . . . . .	253
<b>7</b>	<b>Experimental Features</b>	<b>254</b>
7.1	WARNING . . . . .	254
7.1.1	Node.js bindings . . . . .	254
7.1.2	Navigator Console . . . . .	274
7.1.3	Querying the Navigator local database . . . . .	284
7.1.4	Extractor . . . . .	285
<b>8</b>	<b>Support and Updates</b>	<b>297</b>
8.1	Support and feedback . . . . .	297
8.2	Release notes . . . . .	297
8.2.1	0.11.3 . . . . .	298
8.2.2	0.11.2 . . . . .	298

8.2.3	0.11.1	298
8.2.4	0.11.0	298
8.3	DAML SDK roadmap (as of April 2019)	299

# Chapter 1

## Getting Started

### 1.1 Installing the SDK

#### 1.1.1 1. Install the dependencies

The SDK currently runs on Mac OS or Linux. For Windows support, we recommend installing the SDK on a virtual machine running Linux.

You need to install:

1. [Visual Studio Code](#).
2. [JDK 8 or greater](#).

#### 1.1.2 2. Set up the SDK Assistant

The SDK is distributed via a command-line tool called the [SDK Assistant](#). To install the SDK Assistant:

1. Download the latest installer for your platform:

[Linux installer](#)

[Mac installer](#)

Windows installer coming soon - [sign up to be notified](#)

The SDK is distributed under the Apache 2.0 license ([NOTICES](#)).

2. Run the installer, making sure to follow the instructions to update your PATH:

```
sh ./da-cli-<version>.run
```

3. Run `da setup`, which will download and install the SDK.

#### 1.1.3 3. Configure Maven

To use the Java bindings (and to follow the quickstart guide), you need to install Maven and configure it to use the Digital Asset repository:

1. Install [Maven](#).
2. Download [settings.xml](#).
3. Copy the downloaded file to `~/.m2/settings.xml`. If you already have `~/.m2/settings.xml`, integrate the downloaded file with it instead.

## 1.1.4 Next steps

Follow the [quickstart guide](#).

Read the [introduction](#) page.

Use `da --help` to see all the commands that the SDK Assistant provides.

If you run into any problems, [use the support page](#) to get in touch with us.

## 1.2 Introduction to the product

DAML – Digital Asset’s language for smart contracts – is a new paradigm for thinking about, designing, and writing software for multi-party business processes.

This page explains this paradigm and its benefits.

### 1.2.1 Distributed business processes

No company is an island. All companies need to do business with other companies or consumers. They buy goods, sell services, take on risk, and exchange information. These interactions are underpinned by the legal system and are made up of explicit or implicit contracts that can be enforced by a court of law in the event of a dispute. Such contracts describe *rights* and *obligations* of the parties in a business process.

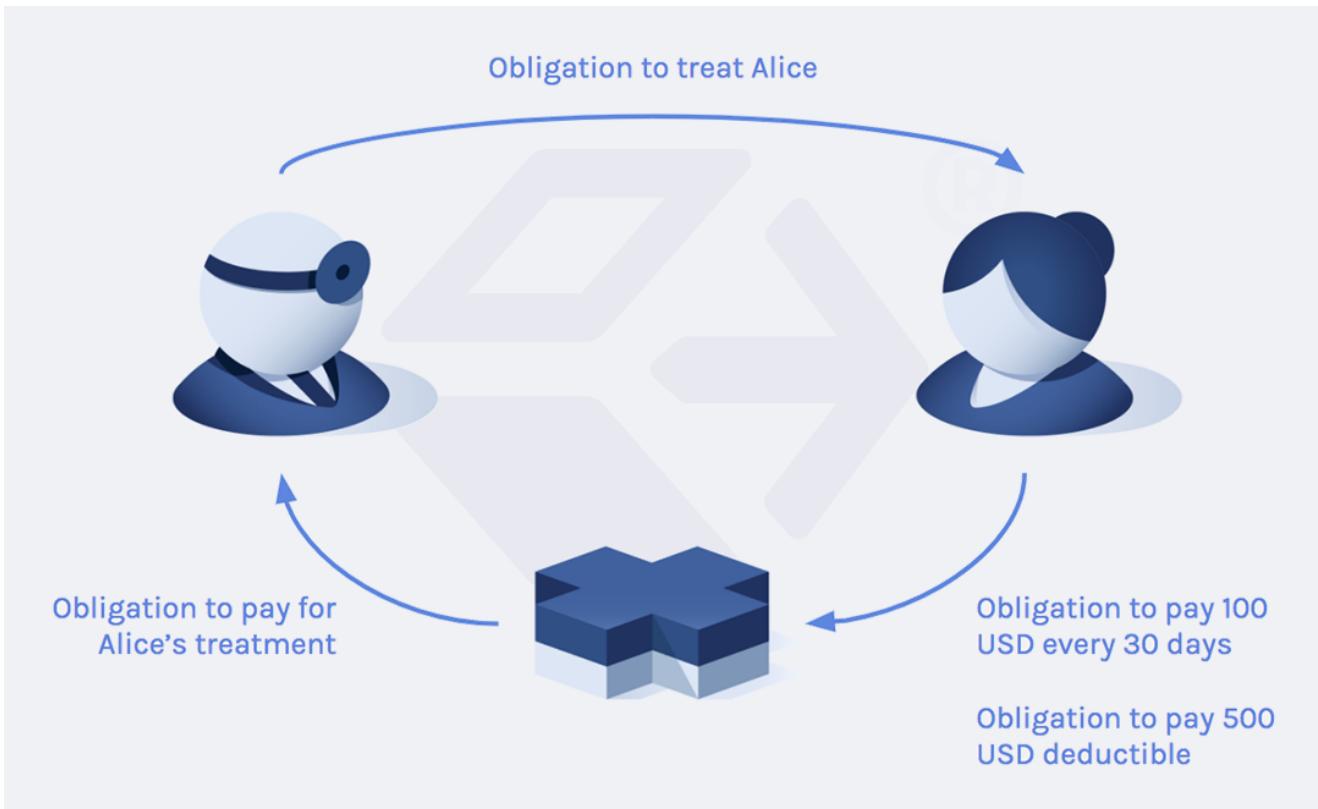
A business process involving independent entities is referred to as a *distributed business process*. To illustrate, consider a simple process defined by the following rights and obligations:

Alice, a consumer, has an obligation to pay her health insurance company a premium of 100 USD every 30 days.

Alice has a right to get treatment from her doctor whenever she wants.

The doctor has a right to get paid by the health insurance company for treating Alice.

Alice has an obligation to pay up to 500 USD of each treatment’s cost to the health insurance company. This is a per-treatment deductible.



We would like to digitize and automate this process so that the following becomes straightforward to achieve:

Alice has an application on her phone that lists her treatments and bills.

The doctor has an application for recording treatments that automatically requests payment from the patient's insurer.

The insurance company automatically pays bills from doctors for insured patients and then sends bills to the patients for their deductibles.

This presents us with the following difficult engineering problems:

We need to guarantee that all parties honor the contracts they have agreed to.

We need to ensure that all the parties collect proofs in the form of digital documents, such that they can enforce their rights in a court of law if a dispute arises.

The privacy of data needs to be preserved. For example, the health insurance company must not have access to all of the doctor's data.

A generalized version of such a process presents further challenges. Alice might switch insurance companies, insurance companies might have re-insurances with each other, Alice could choose to go to any one of thousands of doctors, and there are millions of patients. This becomes increasingly complex because each insurance company will have different data silos, doctors will have different applications for recording their treatments, and data will be stored in a fragmented manner amongst the parties. Such distributed systems are notoriously hard to build such that they work correctly, scale with good performance characteristics, and are resilient.

Today's reality is that distributed business processes often involve many manual steps. Data is entered, shared, and transformed manually because systems lack compatible APIs, data is segregated across different databases, even inside the same company, or the data quality is unreliable and requires human judgement. This leads to slow processing, high cost, high risk of errors, and difficulty providing regulators with consistent and reliable reports.

Digital Asset proposes an alternative model for implementing distributed business processes. One where subject-matter experts supported by software engineers write business processes in a form that is unambiguous and understood by both humans and machines. These processes can then be deployed on the Digital Asset Platform, which is designed to run them at scale while preserving the privacy and integrity of all involved data.

Digital Asset's offering has three main components:

1. DAML for expressing processes in a form that is friendly to both humans and machines.
2. Digital Asset Ledger Server for connecting as a node to the distributed ledger that executes business processes expressed in DAML.
3. DAML SDK for learning about, developing, and testing DAML processes.

## 1.2.2 DAML

DAML is a smart contract language for modeling multi-party business processes that can be deployed on the distributed Digital Asset Platform, which executes these processes with high integrity and privacy guarantees. Financial products and services are examples of such processes that will be used throughout this documentation. DAML is used to express rights and obligations within a computer system. It does not reinvent how business is conducted. Instead, it lets us express contracts in a form that computers can easily work with and that humans can still understand. For example, a DAML encoding of a simple insurance policy for the process described earlier might look like this:

```
-- Copyright (c) 2019 Digital Asset (Switzerland) GmbH and/or its
-- affiliates. All rights reserved.
-- SPDX-License-Identifier: Apache-2.0

daml 1.2
module Insurance where

import DA.Time

template Policy
  with
    patient      : Party
    insurer      : Party
    doctor       : Party
    deductible   : Decimal
    feeAmount    : Decimal
    feeNext      : Time
  where
    ensure deductible > 0.0 && feeAmount > 0.0

    signatory patient, insurer, doctor

    agreement show doctor <> " must treat " <>
                show patient <> " whenever requested. "

    controller insurer can
      InvoiceFee : ContractId Policy
```

(continues on next page)

(continued from previous page)

```

do
    after feeNext
    create Invoice with
        payer      = patient
        receiver   = insurer
        amount     = feeAmount
    create this with
        feeNext   = addRelTime feeNext (days 30)

controller doctor can
    nonconsuming InvoiceTreatment : ()
    with
        amount : Decimal
do
    create Invoice with
        payer      = insurer
        receiver   = doctor
        amount
    create Invoice with
        payer      = patient
        receiver   = insurer
        amount     = min amount deductible
    return ()

template Invoice
with
    payer      : Party
    receiver   : Party
    amount     : Decimal
where
    ensure amount > 0.0

    signatory payer, receiver

    agreement show payer <> " has to pay " <>
        show amount <> " to " <> show receiver <> "."

after : Time -> Update ()
after time = do
    now <- getTime
    assert (time < now)

```

To try out the above code in [DAML Studio](#), you need to add the [DAML Standard Library](#) to your project.

A DAML library describes how parties can interact in a business process using DAML contract templates. DAML contracts encode the rights of the parties as choices that they can exercise, and obligations as agreements that they agree to. DAML does not have any notion of databases, storage schemas, nodes, network addresses, cryptographic keys, or similar concepts. Instead, the secure,

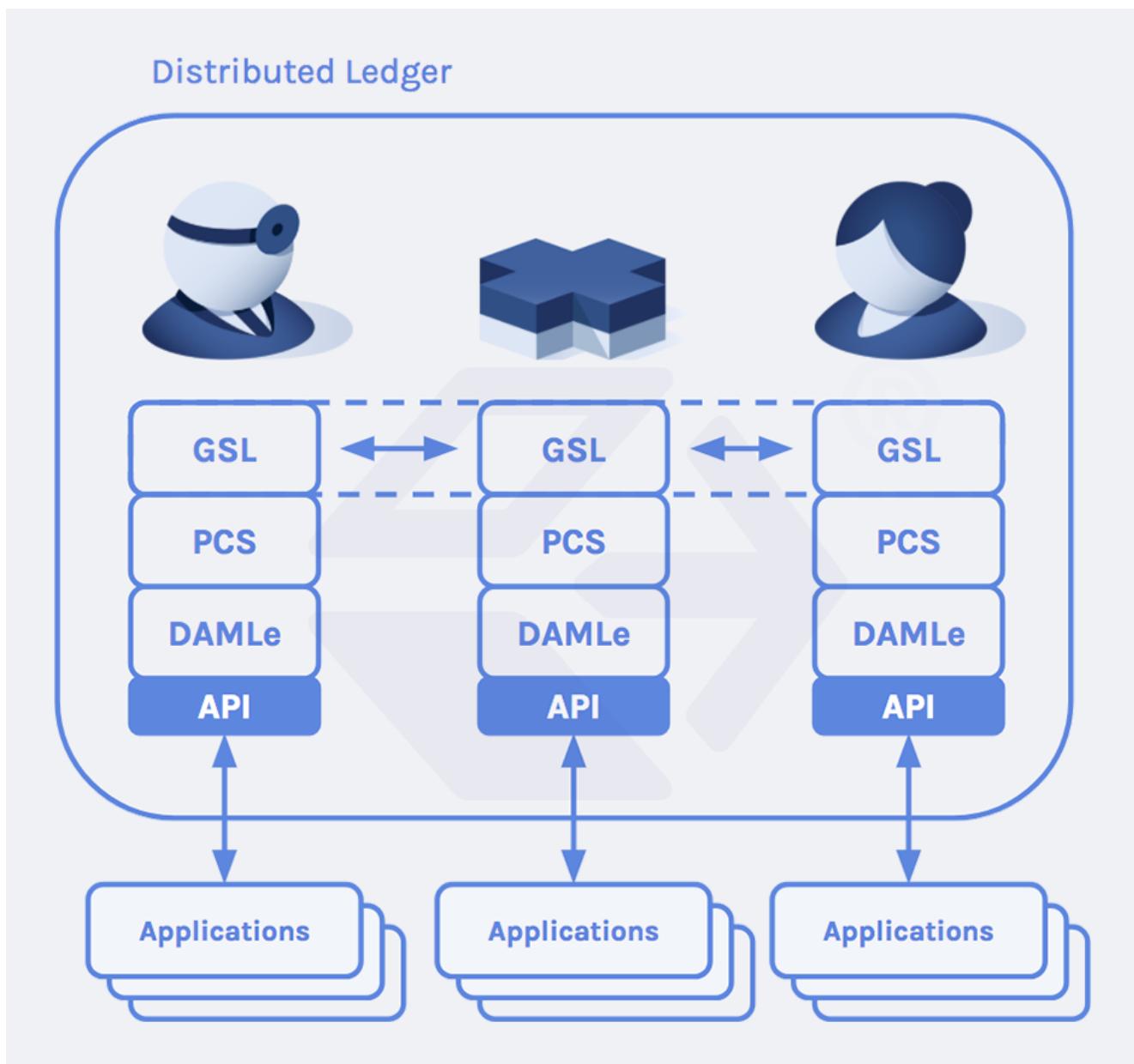
distributed execution of a business process modeled in DAML is performed by the Digital Asset Ledger Server.

DAML is designed to be understood by both humans and machines. This means lawyers can read and assess the meaning of contracts, while developers can write programs that use a party's contracts to, for example, compute risk exposure, cash flow, or customer profiles.

DAML is an advanced, modern language that affords static analysis and strong consistency checks. The DAML SDK includes an Integrated Development Environment (IDE) called DAML Studio that provides feedback about potential problems during development. This makes developing business processes quicker, easier, and safer.

### 1.2.3 Digital Asset Ledger Server

The Digital Asset Ledger Server is the software that runs on the participant nodes that comprise the Digital Asset Ledger. In our example, Alice, the health insurance company, and the doctor would each run a node or delegate the running of their respective nodes to a trusted third party.



A DA Ledger Server interprets DAML code, stores contracts in a Private Contract Store (PCS), and synchronises with other nodes using a Global Synchronization Log (GSL) in a way that is scalable, resilient, and secure. It was built from the ground up with security, data privacy, and performance in mind and it uses state-of-the-art cryptographic primitives for identity keys, hash algorithms, and communication protocols. The DA Ledger Server leverages DAML's precision to:

- Orchestrate business processes between parties
- Enforce that the prescribed business process is followed
- Segregate data on a need-to-know basis among parties, such that each party only gets information about contracts that they are involved in
- Provide cryptographic evidence for the validity of all contracts

The term ledger is derived from the concept of an accounting ledger. Instead of financial transactions, the DA Ledger Server manages contracts.

The DA Ledger Server exposes an API that external applications can use to read contracts, get notified of contract creations and archivals, or exercise contract choices. For example, a smartphone app could use the API to let a user request treatment from a doctor.

DAML libraries are loaded onto the DA Distributed Ledger at runtime, which means that the global system can be adapted easily to support new or updated business processes. Therefore, complex business processes can be modeled, tested, and deployed on the DA Platform much more quickly than is possible with traditional tools.

#### 1.2.4 DAML SDK

The DAML SDK is a collection of tools and documentation for working with and learning about DAML. It includes an IDE for DAML, tools to test your DAML models locally, and learning materials to help you develop the knowledge and skills to implement your own use cases on the Digital Asset Platform.

#### 1.2.5 Summary and next steps

The DAML SDK helps you model distributed business processes in DAML. These can be deployed on the Digital Asset Platform, which offers unique features that allow secure, scalable, and cost-effective automation of complex interactions.

To start using the DAML SDK and DAML go to [Quickstart guide](#) for a guided introduction.

### 1.3 Quickstart guide

In this guide, you will learn about the SDK tools and DAML applications by:

- developing a simple ledger application for issuing, managing, transferring and trading IOUs (I Owe You!)
- developing an integration layer that exposes some of the functionality via custom REST services

Prerequisites:

- You understand what an IOU is. If you are not sure, read the [IOU tutorial overview](#).
- You have installed the DAML SDK. See [Installing the SDK](#).

On this page:

[Download the quickstart application](#)  
- [Folder structure](#)  
[Overview of what an IOU is](#)  
[Run the application using prototyping tools](#)  
[Try out the application](#)  
[Get started with DAML](#)  
- [Develop with DAML Studio](#)  
- [Test using scenarios](#)  
[Integrate with the ledger](#)  
[Next steps](#)

### 1.3.1 Download the quickstart application

You can get the quickstart application as an [SDK template](#):

1. Run `da new quickstart`  
This loads the entire application into a folder called `quickstart`.
2. Run `cd quickstart` to change into the new directory.

#### 1.3.1.1 Folder structure

The project contains the following files:

```
.  
├── da.yaml  
└── daml  
    ├── Iou.daml  
    ├── IouTrade.daml  
    ├── Main.daml  
    └── Tests  
        ├── IouTest.daml  
        └── TradeTest.daml  
├── frontend-config.js  
└── pom.xml  
└── src  
    └── main  
        └── java  
            └── com  
                └── digitalasset  
                    └── quickstart  
                        └── iou  
                            ├── Iou.java  
                            └── IouMain.java  
└── ui-backend.conf
```

`da.yaml` is a DAML project definition file consumed by some CLI commands. It will not be used in this guide.

`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](#) front-end.

`pom.xml` and `src/main/java` constitute a [Java application](#) that provides REST services to interact with the ledger.

You will explore these in more detail through the rest of this guide.

### 1.3.2 Overview of what an IOU is

To run through this guide, you will need to understand what an IOU is. This section describes the properties of an IOU like a bank bill that make it useful as a representation and transfer of value.

A bank bill represents a contract between the owner of the bill and its issuer, the central bank. Historically, it is a bearer instrument - it gives anyone who holds it the right to demand a fixed amount of material value, often gold, from the issuer in exchange for the note.

To do this, the note must have certain properties. In particular, the British pound note shown below illustrates the key elements that are needed to describe money in DAML:



#### 1) The Legal Agreement

For a long time, money was backed by physical gold or silver stored in a central bank. The British pound note, for example, represented a promise by the central bank to provide a certain amount of gold or silver in exchange for the note. This historical artifact is still represented by the following statement:

I promise to pay the bearer on demand the `sum` of five pounds.

The true value of the note comes from the fact that it physically represents a bearer right that is matched by an obligation on the issuer.

#### 2) The Signature of the Counterparty

The value of a right described in a legal agreement is based on a matching obligation for a counter-party. The British pound note would be worthless if the central bank, as the issuer, did not recognize

its obligation to provide a certain amount of gold or silver in exchange for the note. The chief cashier confirms this obligation by signing the note as a delegate for the Bank of England. In general, determining the parties that are involved in a contract is key to understanding its true value.

### 3) The Security Token

Another feature of the pound note is the security token embedded within the physical paper. It allows the note to be authenticated with limited effort by holding it against a light source. Even a third party can verify the note without requiring explicit confirmation from the issuer that it still acknowledges the associated obligations.

## 4) The Unique Identifier

Every note has a unique registration number that allows the issuer to track their obligations and detect duplicate bills. Once the issuer has fulfilled the obligations associated with a particular note, duplicates with the same identifier automatically become invalid.

## 5) The Distribution Mechanism

The note itself is printed on paper, and its legal owner is the person holding it. The physical form of the note allows the rights associated with it to be transferred to other parties that are not explicitly mentioned in the contract.

### 1.3.3 Run the application using prototyping tools

In this section, you will run the quickstart application and get introduced to the main tools for prototyping DAML:

1. To compile the DAML model, run `da run damlc -- package daml/Main.daml target/daml/jou`

This creates a DAR package called target/daml/jou.dar. The output should look like this:

2018-11-21 16:20:50.06 [DEBUG] [package] [] [PA Service Dam]

→ Compiler.Tmp.Handle:2751

Compiling: /.../quickstart/daml/Main.daml

2018-11-21 16:20:50.96 [DEBUG] [package] [] [DA.Service.Dam].

→Compiler.Impl.Handle:2101

Callback received for file Tagged "/.../quickstart/daml/Main.daml"

Created target/daml/qs.dar.

2. To run the `sandbox` (a lightweight local version of the ledger), run `da run sandbox -- --port 7600 --scenario Main:setup target/daml/*`

The output should look like this:

```
Initialized sandbox version 3.0.3 with ledger-id = sandbox-bb4a19ee-  
↳ c0f8-444e-825a-1eb3clecd401, port = 7600, dar file =  
↳ DamlPackageContainer(List(target/daml/qs.dar)), time mode = Static,  
↳ daml-engine = {}
```

(continues on next page)

(continued from previous page)

```
Initialized Static time provider, starting from 1970-01-01T00:00:00Z
DAML LF Engine supports LF versions: 1.0, 0; Transaction versions: 1;
  ↳ Value versions: 1
Starting plainText server
listening on localhost:7600
```

The sandbox is now running, and you can access its [ledger API](#) on port 7600.

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

3. Open a new terminal window and navigate to your project directory.
  4. Start the [Navigator](#), a browser-based ledger front-end, by running `da run navigator --server localhost 7600 --port 7500`
- The Navigator automatically connects the sandbox. You can access it on port 7500.

#### 1.3.4 Try out the application

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

1. Go to <http://localhost:7500/>. This is the Navigator, which you launched [earlier](#).
  2. On the login screen, select **Alice** from the dropdown. This logs you in as Alice. (The list of available parties is specified in the `ui-backend.conf` file.)
- This takes you to the contracts view:

Contracts	ID	Template ID	Time	Choice
#2:2	lou.lou@28b604271fb7...	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, with Id #2:2, created from a template called lou.lou@28b.

3. On the left-hand side, you can see what the pages the Navigator contains:

- Contracts
- Templates
- Owned Ious
- Issued Ious
- Iou Transfers
- Trades

**Contracts** and **Templates** are standard views, available in any application. The others are created just for this application, specified in the `frontend-config.js` file.

For information on creating custom Navigator views, see [Customizable table views](#).

4. Click **Templates** to open the Templates page.

This displays all available contract templates. Instances of contracts (or just contracts) are created from these templates. The names of the templates are of the format `module.template@hash`. Including the hash disambiguates templates, even when identical module and template names are used between packages.

On the far right, you see the number of contract instances that you can see for each template.

5. Try creating a contract from a template. Issue an Iou to yourself by clicking on the `Iou.Iou` row, filling it out as shown below and clicking **Submit**.

6. On the left-hand side, click **Issued Ious** to go to that page. You can see the Iou you just issued yourself.

7. Now, try transferring this Iou to someone else. Click on your Iou, select **Iou\_Transfer**, enter Bob as the new owner and hit **Submit**.

8. Go to the **Owned Ious** page.

The screen shows the same contract #2:2 that you already saw on the Contracts page. It is an Iou for 100, issued by EUR\_Bank.

9. Go to the **Iou Transfers** page. It shows the transfer of your recently issued Iou to Bob, but Bob has not accepted the transfer, so it is not settled.

This is an important part of DAML: nobody can be forced into owning an Iou, or indeed agreeing to any other contract. They must explicitly consent.

You could cancel the transfer by using the `IouTransfer_Cancel` choice within it, but for this walkthrough, leave it alone for the time being.

10. Try asking Bob to exchange your 100 for \$110. To do so, you first have to show your Iou to Bob so that he can verify the settlement transaction, should he accept the proposal.

Go back to **Owned Ious**, open the Iou for 100 and click on the button `Iou_AddObserver`. Submit Bob as the newObserver.

Contracts in DAML are immutable, meaning they can not be changed, only created and archived.

If you head back to the **Owned Ious** screen, you can see that the Iou now has a new Contract ID #6:1.

11. To propose the trade, go to the **Templates** screen. Click on the `IouTrade.IouTrade` template, fill in the form as shown below and submit the transaction.

12. Go to the **Trades** page. It shows the just-proposed trade.

13. You are now going to switch user to Bob, so you can accept the trades you have just proposed. Start by clicking on the logout button next to the username, at the top of the screen. On the login page, select **Bob** from the dropdown.

14. First, accept the transfer of the AliceCoin. Go to the **Iou Transfers** page, click on the row of the transfer, and click **IouTransfer\_Accept**, then **Submit**.

15. Go to the **Owned Ious** page. It now shows the AliceCoin.

It also shows an Iou for \$110 issued by USD\_Bank. This matches the trade proposal you made

Template lou.lou@28b604271fb743f05084198a0fca86c...

observers

[Add new element](#) [Delete last element](#)

issuer

Alice

amount

1.0

currency

AliceCoin

owner

Alice

[Submit](#)

## Template louTrade.louTrade@28b604271fb743f05084198...

baseAmount

100.0

quotelssuer

USD\_Bank

baselssuer

EUR\_Bank

quoteCurrency

USD

baseCurrency

EUR

baselouCid

#6:1

quoteAmount

110.0

buyer

Alice

seller

Bob

Submit

earlier as Alice.

Note its Contract Id #3:2.

16. Settle the trade. Go to the **Trades** page, and click on the row of the proposal. Accept the trade by clicking **IouTrade\_Accept**. In the popup, enter #3:2 as the quoteIouCid, then click **Submit**. The two legs of the transfer are now settled atomically in a single transaction. The trade either fails or succeeds as a whole.

17. Privacy is an important feature of DAML. You can check that Alice and Bob's privacy relative to the Banks was preserved.

To do this, log out, then log in as **USD\_Bank**.

On the **Contracts** page, select **Include archived**. The page now shows all the contracts that **USD\_Bank** has ever known about.

There are just three contracts:

An *IouTransfer* that was part of the scenario during sandbox startup.

Bob's original *Iou* for \$110.

The new \$110 *Iou* owned by Alice. This is the only active contract.

**USD\_Bank** does not know anything about the trade or the EUR-leg. For more information on privacy, refer to the [DA Ledger Model](#).

---

**Note:** **USD\_Bank** does know about an intermediate *IouTransfer* contract that was created and consumed as part of the atomic settlement in the previous step. Since that contract was never active on the ledger, it is not shown in Navigator. You will see how to view a complete transaction graph, including who knows what, in [Test using scenarios](#) below.

---

### 1.3.5 Get started with DAML

The contract model specifies the possible contracts, as well as the allowed transactions on the ledger, and is written in DAML.

The core concept in DAML is a contract template - you used them earlier to create contracts. Contract templates specify:

- a type of contract that may exist on the ledger, including a corresponding data type
- the signatories, who need to agree to the creation of a contract instance of that type
- the rights or choices given to parties by a contract of that type
- constraints or conditions on the data on a contract instance
- additional parties, called observers, who can see the contract instance

For more information about the DA Ledger, consult [DA Ledger Model](#) for an in-depth technical description.

#### 1.3.5.1 Develop with DAML Studio

Take a look at the DAML that specifies the contract model in the quickstart application. The core template is *Iou*.

1. Open [DAML Studio](#), a DAML IDE based on VS Code, by running `da studio` from the root of your project.
2. Using the explorer on the left, open `daml/Iou.daml`.

The first two lines specify language version and module name:

```
daml 1.2
module Iou where
```

Next, a template called `Iou` is declared together with its datatype. This template has five fields:

```
template Iou
with
  issuer : Party
  owner : Party
  currency : Text
  amount : Decimal
  observers : [Party]
```

Conditions for the creation of a contract instance are specified using the `ensure` and `signatory` keywords:

```
ensure amount > 0.0

signatory issuer, owner
```

In this case, there are two conditions:

An `Iou` can only be created if it is authorized by both `issuer` and `owner`.  
The `amount` needs to be positive.

Earlier, as Alice, you authorized the creation of an `Iou`. The `amount` was `100.0`, and Alice as both `issuer` and `owner`, so both conditions were satisfied, and you could successfully create the contract.

To see this in action, go back to the Navigator and try to create the same `Iou` again, but with Bob as `owner`. It will not work.

Observers are specified using the `observer` keyword:

```
observer observers
```

Next, rights or choices are given to `owner`:

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

`controller owner can` starts the block. In this case, `owner` has the right to:

- split the `Iou`
- merge it with another one differing only on `amount`
- initiate a transfer
- add and remove observers

The `Iou_Transfer` choice above takes a parameter called `newOwner` and creates a new `IouTransfer` contract and returns its `ContractId`. It is important to know that, by default, choices consume the contract on which they are exercised. Consuming, or archiving, makes the contract no longer active. So the `IouTransfer` replaces the `Iou`.

A more interesting choice is `IouTrade_Accept`. To look at it, open `IouTrade.daml`.

```
controller seller can
  IouTrade_Accept : (IouId, IouId)
  with
    quoteIouCid : IouId
  do
    baseIou <- fetch baseIouCid
    baseIssuer === baseIou.issuer
    baseCurrency === baseIou.currency
    baseAmount === baseIou.amount
    buyer === baseIou.owner
    quoteIou <- fetch quoteIouCid
    quoteIssuer === quoteIou.issuer
    quoteCurrency === quoteIou.currency
    quoteAmount === quoteIou.amount
    seller === quoteIou.owner
    quoteIouTransferId <- exercise quoteIouCid Iou_Transfer with
      newOwner = buyer
    quoteIouCid <- exercise quoteIouTransferId IouTransfer_Accept
    baseIouTransferId <- exercise baseIouCid Iou_Transfer with
      newOwner = seller
    baseIouCid <- exercise baseIouTransferId IouTransfer_Accept
  return (quoteIouCid, baseIouCid)
```

This choice uses the `==` operator from the [DAML Standard Library](#) to check pre-conditions. The standard library is imported using `import DA.Assert` at the top of the module.

Then, it composes the `Iou_Transfer` and `IouTransfer_Accept` choices to build one big transaction. In this transaction, buyer and seller exchange their ious atomically, without disclosing the entire transaction to all parties involved.

The `Issuers` of the two ious, which are involved in the transaction because they are signatories on the `Iou` and `IouTransfer` contracts, only get to see the sub-transactions that concern them, as we saw earlier.

For a deeper introduction to DAML, consult the [DAML Reference](#).

### 1.3.5.2 Test using scenarios

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

Scenarios are a linear sequence of transactions that is evaluated using the same consistency, conformance and authorization rules as it would be on the full ledger server or the sandbox ledger. They are integrated into DAML Studio, which can show you the resulting transaction graph, making them a powerful tool to test and troubleshoot the contract model.

To take a look at the scenarios in the quickstart application, open `daml/Tests/TradeTest.daml` in DAML Studio.

A scenario test is defined with `trade_test = scenario do`. The `submit` function takes a submitting party and a transaction, which is specified the same way as in contract choices.

The following block, for example, issues an `Iou` and transfers it to Alice:

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

Compare the scenario with the `setup` scenario in `daml/Main.daml`. You will see that the scenario you used to initialize the sandbox is an initial segment of the `trade_test` scenario. The latter adds transactions to perform the trade you performed through Navigator, and a couple of transactions in which expectations are verified.

After a short time, the text `Scenario results` should appear above the test. Click on it to open the visualization of the resulting ledger.

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 (unknown source)
#6:0
└> fetch #5:0 (IouTrade.IouTrade)

#6:1
|   known to (since): 'Bob' (#6), 'Alice' (#6)
└> 'Bob' exercises IouTrade_Accept on #5:0 (IouTrade.IouTrade)
    with
      quoteIouCid = #3:2
  children:
#6:2
|   known to (since): 'Bob' (#6), 'Alice' (#6)
└> fetch #3:2 (Iou.Iou)

#6:3
|   known to (since): 'Bob' (#6), 'Alice' (#6)
└> fetch #3:2 (Iou.Iou)

#6:4
|   known to (since): 'Bob' (#6), 'USD_Bank' (#6), 'Alice' (#6)
└> 'Bob' exercises Iou_Transfer on #3:2 (Iou.Iou)
    with
      newOwner = 'Alice'
  children:
#6:5
|   consumed by: #6:7
|   referenced by #6:6, #6:7
|   known to (since): 'Alice' (#6), 'Bob' (#6), 'USD_Bank' (#6)
└> create Iou.IouTransfer
```

(continues on next page)

(continued from previous page)

```

with
  iou =
    (Iou.Iou with
      issuer = 'USD_Bank';
      owner = 'Bob';
      currency = "USD";
      amount = 110.0;
      observers = []);
  newOwner = 'Alice'

#6:6
| known to (since): 'Bob' (#6), 'Alice' (#6)
└> fetch #6:5 (Iou.IouTransfer)

#6:7
| known to (since): 'Alice' (#6), 'Bob' (#6), 'USD_Bank' (#6)
└> 'Alice' exercises IouTransfer_Accept on #6:5 (Iou.IouTransfer)
    with
    children:
#6:8
|   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 = []

#6:9
| known to (since): 'Bob' (#6), 'Alice' (#6)
└> fetch #4:2 (Iou.Iou)

#6:10
| known to (since): 'Alice' (#6), 'EUR_Bank' (#6), 'Bob' (#6)
└> 'Alice' exercises Iou_Transfer on #4:2 (Iou.Iou)
    with
      newOwner = 'Bob'
    children:
#6:11
|   consumed by: #6:13
|   referenced by #6:12, #6:13
|   known to (since): 'Bob' (#6), 'Alice' (#6), 'EUR_Bank' (#6)
└> create Iou.IouTransfer
    with
      iou =
        (Iou.Iou with
          issuer = 'EUR_Bank';

```

(continues on next page)

(continued from previous page)

```

        owner = 'Alice';
        currency = "EUR";
        amount = 100.0;
        observers = ['Bob']);
newOwner = 'Bob'

#6:12
| known to (since): 'Bob' (#6), 'Alice' (#6)
└> fetch #6:11 (Iou.IouTransfer)

#6:13
| known to (since): 'Bob' (#6), 'Alice' (#6), 'EUR_Bank' (#6)
└> 'Bob' exercises IouTransfer_Accept on #6:11 (Iou.IouTransfer)
    with
    children:
#6:14
| 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/IouTest.daml`, for example, to confirm that the ledger model prevents double spends.

### 1.3.6 Integrate with the ledger

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

To build automations and integrations around the ledger, the SDK has [language bindings](#) for the Ledger API in several programming languages.

To start the Java integration in the quickstart application, run `mvn clean compile exec:java`

The application provides REST services on port 8080 to perform basic operations on behalf of 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 -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 lou with Id ID.

For example, to get the content of the lou with Id 0, run:

```
curl -X GET http://localhost:8080/iou/0
```

PUT on `http://localhost:8080/iou` creates a new lou 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 lou with Id ID.

Find out the Id of your new AliceCoin using step 1. and then run:

```
curl -X POST -d '{ "newOwner":"Bob" }' http://localhost:8080/iou/ID/
transfer
```

to transfer it to Bob.

The automation is based on the [Java bindings](#) and the output of the [Java code generator](#), which are included as a Maven dependency and Maven plugin respectively:

```
<dependency>
    <groupId>com.daml.ledger</groupId>
    <artifactId>bindings-rxjava</artifactId>
    <version>__VERSION__</version>
    <exclusions>
        <exclusion>
            <groupId>com.google.protobuf</groupId>
            <artifactId>protobuf-lite</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

It consists of the application in file `IouMain.java`. It uses the class `Iou` from `Iou.java`, which is generated from the DAML model with the Java code generator. The `Iou` class provides better serialization and de-serialization to JSON via `gson`.

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

```
// create a client object to access services on the ledger
DamlLedgerClient client = DamlLedgerClient.
    ↪forHostWithLedgerIdDiscovery(ledgerhost, ledgerport, Optional.
    ↪empty());

// Connects to the ledger and runs initial validation
client.connect();
```

2. An in-memory contract-store is initialized. This is intended to provide a live view of all active contracts, with mappings between ledger and external Ids.

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

3. The Active Contract 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(e -> Iou.Contract.fromIdAndRecord(e.
        ↪getContractId(), e getArguments()))
            .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.
                ↪fromIdAndRecord(createdEvent.getContractId(), createdEvent.
                ↪getArguments());
                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",
    );
}

```

(continues on next page)

(continued from previous page)

```

    UUID.randomUUID().toString(),
    party,
    Instant.EPOCH,
    Instant.EPOCH.plusSeconds(10),
    Collections.singletonList(c))
    .blockingGet();
}

```

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

Listing 1: `ExerciseCommand`

```

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

```

Listing 2: `CreateCommand`

```

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

```

The rest of the application sets up the REST services using [Spark Java](#), and does dynamic package Id detection using the Package Service. The latter is useful during development when package Ids change frequently.

For a discussion of ledger application design and architecture, take a look at [Application Architecture Guide](#).

### 1.3.7 Next steps

Great - you've completed the quickstart guide!

Some steps you could take next include:

Explore [examples](#) for guidance and inspiration.

[Learn DAML](#).

[Learn more about application development](#).

[Learn about the conceptual models](#) behind DAML and platform.

## Chapter 2

# Modeling Processes with DAML

## 2.1 DAML reference docs

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

### 2.1.1 Overview: template structure

This page covers what a template looks like: what parts of a template there are, and where they go.

For the structure of a DAML file outside a template, see [Reference: DAML file structure](#).

Template outline structure  
Choice structure  
Choice body structure

#### 2.1.1.1 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
```

(continues on next page)

(continued from previous page)

**True**

```
controller exampleParty3 can
  -- a choice goes here; see next section
```

**template name** template keyword**parameters** with followed by the names of parameters and their types**template body** where keyword

Can include:

**signatories** signatory keywordThe parties (see the [Party](#) type) who must consent to the creation of an instance of this contract.**observers** observer keyword

Parties that aren't signatories but can still see this contract.

**an agreement** agreement keyword

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.

**choices** controller nameOfParty can nameOfChoiceDefines choices that can be exercised. See [Choice structure](#) for what can go in a choice.

### 2.1.1.2 Choice structure

Here's the structure of a choice inside a template:

```
controller exampleParty3 can
  NameOfChoice : -- return type here
    ()
  with
    -- parameters here
    party : Party
  do
    -- choice body; see next section
  return ()
```

**a controller (or controllers)** controller keyword

Who can exercise the choice.

**consumability** nonconsuming keyword

By default, contracts are archived when a choice on them is exercised. If you include nonconsuming, this choice can be exercised over and over.

**a name** Must begin with a capital letter. Must be unique - choices in different templates can't have the same name.**a return type** after a :, the return type of the choice**choice arguments** with keyword**a choice body** After do keywordWhat happens when someone exercises the choice. A choice body can contain update statements: see [Choice body structure](#) below.

### 2.1.1.3 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
```

**abort** Stop execution of the choice, fail the update.

```
if False then abort
```

**assert** Fail the update unless the condition is true. Usually used to limit the arguments that can be supplied to a contract choice.

```
assert (amount > 0)
```

**getTime** Gets the ledger effective time. Usually used to restrict when a choice can be exercised.

```
currentTime <- getTime
```

**return** Explicitly return a value. By default, a choice returns the result of its last update expression. This means you only need to use return if you want to return something else.

```
return amount
```

The choice body can also contain:

**let keyword** Used to assign values or functions.

**assign a value to the result of an update statement** For example: `contractFetched <- fetch someContractId`

### 2.1.2 Reference: templates

This page gives reference information on templates:

Template name
Template parameters
Signatory parties
Observers
Choices
Agreements
Preconditions

For the structure of a template, see [Overview: template structure](#).

#### 2.1.2.1 Template name

<code>template NameOfTemplate</code>
--------------------------------------

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).

### 2.1.2.2 Template parameters

```
template NameOfTemplate
with
exampleParty : Party
exampleParty2 : Party
exampleParty3 : Party
-- 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. You must pass in the parties as parameters to the contract.

This means isn't valid to replace the `giver` variable in the `Payout` template above directly with 'Elizabeth'.

Parties can only be named explicitly in scenarios.

### 2.1.2.3 Signatory parties

```
where
signatory exampleParty
```

`signatory` keyword. After `where`. Followed by at least one `Party`.

Signatories are the parties (see the `Party` type) who must consent to the creation of an instance of this contract. They are the parties who would be put into an *obligable* position when this contract is created.

DAML won't let you put someone into an obligable position without their consent. So if the contract will cause obligations for a party, they must be a signatory.

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).

You must have least one signatory per template. You can have many, either as a comma-separated list or reusing the keyword.

### 2.1.2.4 Observers

```
where
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.

Use when a party needs visibility on a contract, or be informed of contract events, but is not a `signatory` or `controller`.

### 2.1.2.5 Choices

```
where
```

```
  controller exampleParty can  
  -- a choice goes here; see next page
```

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.

See [Reference: choices](#) for full reference information.

### 2.1.2.6 Agreements

```
where
```

```
  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 `<>`.

### 2.1.2.7 Preconditions

```
where
```

```
  ensure  
  -- boolean condition
```

`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.

## 2.1.3 Reference: choices

This page gives reference information on choices:

Controllers  
 Choice name  
 Non-consuming choices  
 Return type  
 Choice arguments  
 Choice body

For the structure of a choice, see [Overview: template structure](#).

### 2.1.3.1 Controllers

```
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.

### 2.1.3.2 Choice name

```
controller exampleParty can  
NameOfChoice
```

The name of the choice. Must begin with a capital letter.

Must be unique in your project. Choices in different templates can't have the same name. You can have multiple choices after one **can**, for tidiness.

### 2.1.3.3 Non-consuming choices

```
controller exampleParty can  
nonconsuming NameOfChoice2
```

nonconsuming keyword. Optional.

Makes a choice non-consuming: that is, exercising the choice does not archive the contract. By default, choices are *consuming*: when a choice on a contract is exercised, that contract instance is archived. Archived means that it's permanently marked as being inactive, and no more choices can be exercised on it, though it still exists on the ledger.

This is useful in the many situations when you want to be able to exercise a choice more than once.

### 2.1.3.4 Return type

```
controller exampleParty can  
NameOfChoice4 : ContractId NameOfTemplate
```

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.

### 2.1.3.5 Choice arguments

```
controller exampleParty can
  NameOfChoice3 : ExampleReturnType
    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.

### 2.1.3.6 Choice body

```
controller exampleParty can
  NameOfChoice4 : ContractId NameOfTemplate
    do
      create NameOfTemplate with exampleParty; exampleParty2;
    ↪exampleParty3
```

Introduced with do

The logic in this section is what is executed when the choice gets exercised.

The choice body contains Update expressions. For detail on this, see [Reference: updates](#).

By default, the last expression in the choice is returned. You can return multiple updates in tuple form or in a custom data type. To return something that isn't of type Update, use the return keyword.

## 2.1.4 Reference: updates

This page gives reference information on Updates:

```
Background
Binding variables
do
create
exercise
fetch
abort
assert
getTime
return
let
```

this

For the structure around them, see [Overview: template structure](#).

#### 2.1.4.1 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](#).

#### 2.1.4.2 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.

#### 2.1.4.3 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.

#### 2.1.4.4 create

`create NameOfTemplate with exampleParty; exampleParty2; exampleParty3`

`create` keyword.

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](#).

#### 2.1.4.5 exercise

```
exercise IdOfContract NameOfChoiceOnContract with choiceArgument1 = value1
```

exercise keyword.

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.

#### 2.1.4.6 fetch

```
fetchedContract <- fetch IdOfContract
```

fetch keyword.

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.

#### 2.1.4.7 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.

#### 2.1.4.8 assert

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

assert keyword.

Fails the transaction if the condition is false. So the choice can only be exercised if the boolean expression evaluates to True.

Often used to restrict the arguments that can be supplied to a contract choice.

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

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

### 2.1.4.9 getTime

```
currentTime <- getTime
```

getTime keyword.

Gets the ledger effective time. (You will usually want to immediately bind it to a variable in order to be able to access the value.)

Used to restrict when a choice can be made. For example, with an assert that the time is later than a certain time.

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

```
Complete : ()  
  
do  
  -- bind the ledger effective time to the tchoose variable using  
  ↵getTime  
  tchoose <- getTime
```

### 2.1.4.10 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)
```

### 2.1.4.11 let

See the documentation on [Let](#).

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

```
do  
  -- defines a function, createdContract, taking a single argument that  
  ↵when  
  -- called _will_ create the new contract using argument for issuer and  
  ↵owner  
  let createContract x = create NameOfContract with issuer = x; owner = x  
  
  createContract party1  
  createContract party2
```

### 2.1.4.12 this

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

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

## 2.1.5 Reference: data types

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

### Built-in types

- [Table of built-in primitive types](#)
- [Escaping characters](#)
- [Time](#)

### Lists

- [Summing a list](#)

### Records and record types

- [Data constructors](#)
- [Accessing record fields](#)
- [Updating record fields](#)
- [Parameterized data types](#)

### Type synonyms

- [Function types](#)

### Algebraic data types

- [Product types](#)
- [Sum types](#)
- [Pattern matching](#)

### 2.1.5.1 Built-in types

## Table of built-in primitive types

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

## Escaping characters

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

## Time

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

### 2.1.5.2 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.

### 2.1.5.3 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.

### 2.1.5.4 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`.

### 2.1.5.5 Algebraic data types

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

#### Product types

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

To get around this, wrap the values in a [record](#): `data Coordinate a b = Coordinate {first: a; second: b}`.

These kinds of types are called product types.

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

#### Sum types

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

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

A very useful sum type is `data Optional a = None | Some a`. It is part of the [DAML standard library](#).

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

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

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

#### Pattern matching

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

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

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

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

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

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

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

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

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

Notice the use of nested pattern matching above.

---

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

---

## 2.1.6 Reference: built-in functions

This page gives reference information on functions for:

- [Working with time](#)
- [Working with numbers](#)
- [Working with text](#)
- [Working with lists](#)
  - [Folding](#)

### 2.1.6.1 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`: adds times. Takes a Time and RelTime and adds the RelTime to the Time.  
`days, hours, minutes, seconds`: constructs a RelTime of the specified length.  
`pass`: (in scenario tests only) use `pass : RelTime -> Scenario` Time to advance the ledger effective time by the argument amount. Returns the new time.

### 2.1.6.2 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.

### 2.1.6.3 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

#### 2.1.6.4 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)
```

## 2.1.7 Reference: expressions

This page gives reference information for DAML expressions that are not [updates](#):

### Definitions

- [Values](#)
- [Functions](#)

### Arithmetic operators

### Comparison operators

### Logical operators

### If-then-else

### Let

### 2.1.7.1 Definitions

Use assignment to bind values or functions at the top level of a DAML file or in a contract template body.

#### Values

For example:

```
pi = 3.1415926535
```

The fact that `pi` has type `Decimal` is inferred from the value. To explicitly annotate the type, mention it after a colon following the variable name:

```
pi : Decimal = 3.1415926535
```

#### Functions

You can define functions. Here's an example: a function for computing the surface area of a tube:

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

Here you see:

the name of the function

the function's type signature `Decimal -> Decimal -> Decimal`

This means it takes two Decimals and returns another Decimal.

the definition = `2.0 * pi * r * h` (which uses the previously defined `pi`)

### 2.1.7.2 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.

### 2.1.7.3 Comparison operators

Operator	Works for
<, <=, >, >=	Bool, Text, Int, Decimal, Party, Time
==, /=	Bool, Text, Int, Decimal, Party, Time, and <a href="#">identifiers of contract instances</a> stemming from the same contract template

### 2.1.7.4 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.

### 2.1.7.5 If-then-else

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

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

### 2.1.7.6 Let

To bind values or functions to be in scope beneath the expression, use the block keyword let:

```
doubled =
-- let binds values or functions to be in scope beneath the expression
let
  double (x : Int) = 2 * x
  up = 5
in double up
```

You can use let inside do and scenario blocks:

```
blah = scenario
do
  let
    x = 1
    y = 2
    -- x and y are in scope for all subsequent expressions of the do
  ↪block,
    -- so can be used in expression1 and expression2.
  expression1
  expression2
```

Lastly, a template may contain a single let block.

```
template IoU
with
  issuer : Party
  owner : Party
where
  signatory issuer

  let updateOwner o = create this with owner = o
    updateAmount a = create this with owner = a

    -- Expressions bound in a template let block can be referenced
    -- from any and all of the signatory, consuming, ensure and
    -- agreement expressions and from within any choice do blocks.

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

## 2.1.8 Reference: functions

This page gives reference information on functions in DAML:

- [Defining functions](#)
- [Partial application](#)
- [Functions are values](#)

## Generic functions

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

### 2.1.8.1 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  $\rightarrow$  as:

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

### 2.1.8.2 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
```

### 2.1.8.3 Functions are values

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

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

pi : Decimal = 3.1415926535

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

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

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

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

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

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

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

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

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

The [Folding](#) section looks into two useful built-in functions, `foldl` and `foldr`, that also take a function as an argument.

---

**Note:** DAML does not allow functions as parameters of contract templates and contract choices. However, a follow up of a choice can use built-in functions, defined at the top level or in the contract template body.

### 2.1.8.4 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.

---

### 2.1.9 Reference: scenarios

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

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

For an introduction to scenarios, see [Testing DAML using scenarios](#).

#### 2.1.9.1 Scenario keyword

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

#### 2.1.9.2 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 ...`

#### 2.1.9.3 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 ...
```

#### 2.1.9.4 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).

#### 2.1.10 Reference: DAML file structure

This page gives reference information on the structure of DAML files outside of `templates`:

<a href="#">File structure</a>
<a href="#">Imports</a>
<a href="#">Libraries</a>
<a href="#">Comments</a>
<a href="#">Contract identifiers</a>

### 2.1.10.1 File structure

Language version (`daml 1.2`).

This file's module name (`module NameOfFile 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`.

### 2.1.10.2 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.

### 2.1.10.3 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.

Bear in mind that the `LibraryModules.daml` files are discovered by a bottom up search from the directories of each of the open DAML files. The search uses the first instance of `LibraryModules.daml` it finds.

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.

### 2.1.10.4 Comments

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

### 2.1.10.5 Contract identifiers

When an instance of a template (that is, a contract) is added to the ledger, it's assigned a unique identifier, of type `ContractId <name of template>`.

The runtime representation of these identifiers depends on the execution environment: a contract identifier from the Sandbox looks different to one on the DA Platform.

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

## 2.2 DAML Standard Library

The DAML Standard Library is a collection of DAML modules organised in domains that can be used to implement concrete applications.

### 2.2.1 Usage

The standard library is included in the DAML compiler so it can be used straight out of the box. You can import modules from the standard library just like your own, e.g.:

```
import DA.Optional
```

### 2.2.2 Domains

#### 2.2.2.1 Base for DAML 1.2

The base domain of the DAML standard library contains various modules with common definitions and helper functions.

##### Module Prelude

##### Typeclasses

##### class Functor f where

**fmap** : (a → b) → f a → f b

**(<\$)** : a → f b → f a

Replace all locations in the input with the same value. The default definition is `fmap . const`, but this may be overridden with a more efficient version.

**class Bounded a where** The `Bounded` class is used to name the upper and lower limits of a type. `Ord` is not a superclass of `Bounded` since types that are not totally ordered may also have upper and lower bounds.

The `Bounded` class may be derived for any enumeration type; `minBound` is the first constructor listed in the data declaration and `maxBound` is the last. `Bounded` may also be derived for single-constructor datatypes whose constituent types are in `Bounded`.

**minBound** : a

**maxBound** : a

**class Enum a where** Class `Enum` defines operations on sequentially ordered types.

The `enumFrom` methods are used in Haskell's translation of arithmetic sequences.

Instances of `Enum` may be derived for any enumeration type (types whose constructors have no fields). The nullary constructors are assumed to be numbered left-to-right by `fromEnum` from 0 through `n-1`. See Chapter 10 of the /Haskell Report/ for more details.

For any type that is an instance of class `Bounded` as well as `Enum`, the following should hold:

The calls `succ maxBound` and `pred minBound` should result in a runtime error.

`fromEnum` and `toEnum` should give a runtime error if the result value is not representable in the result type. For example, `toEnum 7 :: Bool` is an error.

`enumFrom` and `enumFromThen` should be defined with an implicit bound, thus:

```

enumFrom      x = enumFromTo      x maxBound
enumFromThen x y = enumFromThenTo x y bound
where
    bound | fromEnum y >= fromEnum x = maxBound
           | otherwise                  = minBound

succ : a -> a
the successor of a value. For numeric types, ‘succ’ adds 1.
pred : a -> a
the predecessor of a value. For numeric types, ‘pred’ subtracts 1.
toEnum : Int -> a
Convert from an ‘Int’.
fromEnum : a -> Int
Convert to an ‘Int’. It is implementation-dependent what ‘fromEnum’ returns when applied to a value that is too large to fit in an ‘Int’.
enumFrom : a -> [a]
Used in Haskell’s translation of [n..] with [n..] = enumFrom n, a possible implementation being enumFrom n = n : enumFrom (succ n). For example:
    enumFrom 4 :: [Integer] = [4,5,6,7,...]
    enumFrom 6 :: [Int] = [6,7,8,9,...,maxBound :: Int]
enumFromThen : a -> a -> [a]
Used in Haskell’s translation of [n,n'..] with [n,n'..] = enumFromThen n n', a possible implementation being enumFromThen n n' = n : n' : worker (f x) (f x n'), worker s v = v : worker s (s v), x = fromEnum n' - fromEnum n and f n y | n > 0 = f (n - 1) (succ y) | n < 0 = f (n + 1) (pred y) | otherwise = y For example:
    enumFromThen 4 6 :: [Integer] = [4,6,8,10...]
    enumFromThen 6 2 :: [Int] = [6,2,-2,-6,...,minBound :: Int]
enumFromTo : a -> a -> [a]
Used in Haskell’s translation of [n..m] with [n..m] = enumFromTo n m, a possible implementation being enumFromTo n m | n <= m = n : enumFromTo (succ n) m | otherwise = []. For example:
    enumFromTo 6 10 :: [Int] = [6,7,8,9,10]
    enumFromTo 42 1 :: [Integer] = []
enumFromThenTo : a -> a -> a -> [a]
Used in Haskell’s translation of [n,n'..m] with [n,n'..m] = enumFromThenTo n n' m, a possible implementation being enumFromThenTo n n' m = worker (f x) (c x) n m, x = fromEnum n' - fromEnum n, c x = bool (>=) (<=) (x > 0) f n y | n > 0 = f (n - 1) (succ y) | n < 0 = f (n + 1) (pred y) | otherwise = y and worker s c v m | c v m = v : worker s c (s v) m | otherwise = [] For example:
    enumFromThenTo 4 2 -6 :: [Integer] = [4,2,0,-2,-4,-6]
    enumFromThenTo 6 8 2 :: [Int] = []

class Additive a where Basic additive class. Instances are expected to respect the following laws:
Associativity of (+): (x + y) + z = x + (y + z)
Commutativity of (+): x + y = y + x
x + aunit = x
negate gives the additive inverse: x + negate x = aunit
(+) : a -> a -> a
(-) : a -> a -> a
negate : a -> a

```

```
aunit : a
```

**class Multiplicative a where** Basic multiplicative class. Instances are expected to respect the following laws:

- Associativity of (\*)**:  $(x * y) * z = x * (y * z)$
- Commutativity of (\*)**:  $x * y = y * x$
- $x * \text{one} = x$
- $(*) : a \rightarrow a \rightarrow a$
- munit** : a

**class (Additive a, Multiplicative a) => Number a where** Basic numeric class. Instances are usually expected to respect the following law (in addition to laws from being additive and multiplicative)

- Distributivity of (\*) with respect to (+)**:  $a * (b + c) = (a * b) + (a * c)$  and  $(b + c) * a = (b * a) + (c * a)$

**class Signed a where**

```
signum : a -> a
```

```
abs : a -> a
```

Absolute value.

**class Show a where** Conversion of values to readable ‘String’s.

Derived instances of ‘Show’ have the following properties, which are compatible with derived instances of ‘Text.Read.Read’:

The result of ‘show’ is a syntactically correct Haskell expression containing only constants, given the fixity declarations in force at the point where the type is declared. It contains only the constructor names defined in the data type, parentheses, and spaces. When labelled constructor fields are used, braces, commas, field names, and equal signs are also used. If the constructor is defined to be an infix operator, then ‘showsPrec’ will produce infix applications of the constructor.

The representation will be enclosed in parentheses if the precedence of the top-level constructor in  $x$  is less than  $d$  (associativity is ignored). Thus, if  $d$  is 0 then the result is never surrounded in parentheses; if  $d$  is 11 it is always surrounded in parentheses, unless it is an atomic expression.

If the constructor is defined using record syntax, then ‘show’ will produce the record-syntax form, with the fields given in the same order as the original declaration.

For example, given the declarations

```
infixr 5 :^:
data Tree a = Leaf a | Tree a :^: Tree a
```

the derived instance of ‘Show’ is equivalent to

```
instance (Show a) => Show (Tree a) where
    showsPrec d (Leaf m) = showParen (d > app_prec) $
        showString "Leaf " . showsPrec (app_prec+1) m
    where app_prec = 10

    showsPrec d (u :^: v) = showParen (d > up_prec) $
        showsPrec (up_prec+1) u .
        showString " :^: " .
        showsPrec (up_prec+1) v
    where up_prec = 5
```

Note that right-associativity of `:^:` is ignored. For example,

`show (Leaf 1 :^: Leaf 2 :^: Leaf 3)` produces the string "Leaf 1 :^: (Leaf 2 :^: Leaf 3)".

**showsPrec** : Int → a → ShowS

Convert a value to a readable String.

`showsPrec` should satisfy the law

```
showsPrec d x r ++ s == showsPrec d x (r ++ s)
```

Derived instances of 'Text.Read.Read' and 'Show' satisfy the following:

`(x, "")` is an element of `(Text.Read.readsPrec d (showsPrec d x ""))`.

That is, `Text.Read.readsPrec` parses the string produced by `showsPrec`, and delivers the value that `showsPrec` started with.

**show** : a → Text

A specialised variant of 'showsPrec', using precedence context zero, and returning an ordinary 'String'.

**showList** : [a] → ShowS

The method 'showList' is provided to allow the programmer to give a specialised way of showing lists of values. For example, this is used by the predefined 'Show' instance of the 'Char' type, where values of type 'String' should be shown in double quotes, rather than between square brackets.

**class HasTime m where** Places where the time is available - basically Scenario and Update.

**getTime** : m Time

Obtain the current time.

**class (Action m) => CanAbort m where**

**abort** : Text → m a

Abort the current action with a message.

**class (Multiplicative a) => Fractional a where**

**(/)** : a → a → a

x / y divides x by y.

**recip** : a → a

Calculate the reciprocal: `recip x` is  $1/x$ .

**class (Functor f) => Applicative f where**

**pure** : a → f a

Lift a value.

**(<\*>)** : f (a → b) → f a → f b

Sequential application.

A few functors support an implementation of '<\*>' that is more efficient than the default one.

**liftA2** : (a → b → c) → f a → f b → f c

Lift a binary function to actions.

Some functors support an implementation of 'liftA2' that is more efficient than the default one. In particular, if 'fmap' is an expensive operation, it is likely better to use 'liftA2' than to 'fmap' over the structure and then use '<\*>'.

**(\*>)** : f a → f b → f b

Sequence actions, discarding the value of the first argument.

---

**(<\*)** : f a -> f b -> f a  
Sequence actions, discarding the value of the second argument.

**class (Applicative m) => Action m where**

**(>>=)** : m a -> (a -> m b) -> m b  
Sequentially compose two actions, passing any value produced by the first as an argument to the second.

**class (Action m) => ActionFail m where**

**fail** : Text -> m a  
Fail with an error message.

**class Semigroup a where** The class of semigroups (types with an associative binary operation).  
**(<>)** : a -> a -> a  
An associative operation.

**class (Semigroup a) => Monoid a where** The class of monoids (types with an associative binary operation that has an identity).  
**mempty** : a  
Identity of (<>)  
**mconcat** : [a] -> a  
Fold a list using the monoid. For example using mconcat on a list of strings would concatenate all strings to one lone string.

**class Template c where**

**ensure** : c -> Bool  
Predicate that must hold for the successful creation of the contract.

**signatory** : c -> [Party]  
The signatories of a contract.

**observer** : c -> [Party]  
The observers of a contract.

**agreement** : c -> Text  
The agreement text of a contract.

**class (Template c) => Choice c e r where**

**choiceController** : c -> e -> [Party]

**choice** : c -> ContractId c -> e -> Update r

**class IsParties a where** Accepted ways to specify a list of parties: either a single party, or a list of parties.  
**toParties** : a -> [Party]  
Convert to list of parties.

## Data Types

**type Shows**  
= Text -> Text  
The shows functions return a function that prepends the output ‘String’ to an existing ‘String’. This allows constant-time concatenation of results using function composition.

**data [] a**

Documentation for lists

[]

: a [a]

**data Bool**

**False**

**True**

**data Constraint**

The kind of constraints, like Show a

**data Decimal**

**data Int**

A 64-bit integer.

**data Ordering**

Information about ordering

**LT**

**EQ**

**GT**

**data Proxy a**

**Proxy**

**data Symbol**

(Kind) This is the kind of type-level symbols. Declared here because class IP needs it

**data Text**

**data ContractId a**

The ContractId a type represents an id for a contract made from template a. The id can be used to fetch the contract, among other things.

**ContractId Opaque**

**data Date**

The Date type represents a date.

**Date Opaque**

**data Party**

The Party type represents a party to a contract.

**Party Opaque**

**data Scenario a**

The `Scenario` type is used to simulate multi-party ledger interactions. The type `Scenario a` describes a set of actions taken by various parties during the simulated scenario, before returning a value of type `a`.

### **Scenario Opaque**

`data Time`

The `Time` type represents a specific datetime in UTC, i.e. a date and a time in UTC.

### **Time Opaque**

`data Update a`

The `Update a` type represents an action to update or query the ledger, before returning a value of type `a`.

### **Update Opaque**

`data Down a`

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

### **Down a**

`data Either a b`

The `Either` type represents values with two possibilities: a value of type `Either a b` is either `Left a` or `Right b`.

The `Either` type is sometimes used to represent a value which is either correct or an error; by convention, the `Left` constructor is used to hold an error value and the `Right` constructor is used to hold a correct value (mnemonic: right also means correct).

### **Left a**

### **Right b**

`data Optional a`

The `Optional` type encapsulates an optional value. A value of type `Optional a` either contains a value of type `a` (represented as `Some a`), or it is empty (represented as `None`). Using `Optional` is a good way to deal with errors or exceptional cases without resorting to drastic measures such as `error`.

The `Optional` type is also a monad. It is a simple kind of error monad, where all errors are represented by `None`. A richer error monad can be built using the `Data.Either.Either` type.

### **None**

### **Some a**

`data Archive`

### **Archive**

`data ChoiceType`

### **Consuming**

### **NonConsuming**

## Functions

**otherwise** : Bool

**getTag** : a -> Int#

**(++)** : Text -> Text -> Text

**map** : (a -> b) -> [a] -> [b]

**foldr** : (a -> b -> b) -> b -> [a] -> b

**(.)** : (b -> c) -> (a -> b) -> a -> c

Function composition.

**const** : a -> b -> a

const x is a unary function which evaluates to x for all inputs.

```
>>> const 42 "hello"  
42
```

```
>>> map (const 42) [0..3]  
[42, 42, 42, 42]
```

**eftInt** : Int -> Int -> [Int]

**efdInt** : Int -> Int -> [Int]

**efdtInt** : Int -> Int -> Int -> [Int]

**efdtIntUp** : Int -> Int -> Int -> [Int]

**go\_up** : Int -> Int -> Int -> [Int]

**efdtIntDn** : Int -> Int -> Int -> [Int]

**go\_dn** : Int -> Int -> Int -> [Int]

**error** : Text -> a

**showList\_\_** : (a -> ShowS) -> [a] -> ShowS

**showI** : (a -> ShowS) -> [a] -> ShowS

**showParen** : Bool -> ShowS -> ShowS

utility function that surrounds the inner show function with parentheses when the ‘Bool’ parameter is ‘True’.

**showString** : Text -> ShowS

utility function converting a ‘String’ to a show function that simply prepends the string unchanged.

**showSpace** : ShowS

**showCommaSpace** : ShowS

**assert** : (CanAbort m) => Bool -> m ()

Check whether a condition is true, and otherwise abort.

**assertMsg** : (CanAbort m) => Text -> Bool -> m ()

Check whether a condition is true, and otherwise abort with a message.

**assertAfter** : (CanAbort m, HasTime m) => Time -> m ()

Check whether the given time is in the future, and otherwise abort.

**assertBefore** : (CanAbort m, HasTime m) => Time -> m ()

Check whether the given time is in the past, and otherwise abort.

**daysSinceEpochToDate** : Int -> Date

Convert number of days since epoch (i.e. the number of days since January 1, 1970) to a date.

**dateToDaysSinceEpoch** : Date -> Int

Convert a date to number of days epoch (i.e. the number of days since January 1, 1970).

**partyToText** : Party -> Text

Convert the Party to Text, giving back what you passed to getParty. Most users should stick to show of the party instead, which also wraps the party in 'ticks' making it clear it was a Party originally.

**partyFromText** : Text -> Optional Party

Converts a Text to Party. It returns None if the provided text contains any forbidden characters. See DAML-LF spec for a specification on which characters are allowed in parties. Note that this function accepts text without single quotes.

Also note that this function does not perform any check on whether the provided text correspond to a party that exists on a given ledger: it merely converts the given Text to a Party. The only way to guarantee that a given Party exists on a given ledger is to involve it in a contract.

This function, together with partyToText, forms an isomorphism between valid party strings and parties. In other words, the following equations hold:

```
| p. partyFromText (partyToText p) = Some p
| txt p. partyFromText txt = Some p ==> partyToText p = txt
```

This function will crash at runtime if you compile DAML to DAML-LF < 1.2.

**getParty** : Text -> Scenario Party

Get the party with the given name. Party names must be non-empty and only contain alphanumeric characters, space, - (dash) or \_ (underscore).

**submit** : Party -> Update a -> Scenario a

submit p u describes the scenario in which party p attempts to update the ledger with update action u, and returns the value returned by the underlying update action. This scenario is considered a failure if the underlying update action fails.

**submitMustFail** : Party -> Update a -> Scenario ()

submitMustFail describes the scenario in which party p attempts to update the ledger with update action u, but is expected to fail. This scenario is considered a failure if the underlying update action succeeds.

**scenario** : Scenario a -> Scenario a

Declare you are building a scenario.

**(\\$)** : (a -> b) -> a -> b

**curry** : ((a, b) -> c) -> a -> b -> c

Turn a function that takes a pair into a function that takes two arguments.

**uncurry** : (a -> b -> c) -> (a, b) -> c

Turn a function that takes two arguments into a function that takes a pair.

**( $\wedge$ )** : Int -> Int -> Int  
x  $\wedge$  n raises x to the power of n.

**(%)** : Int -> Int -> Int  
x % y calculates the remainder of x by y

**(>>)** : (Applicative m) => m a -> m b  
Sequentially compose two actions, discarding any value produced by the first, like sequencing operators (such as the semicolon) in imperative languages.

**ap** : (Applicative f) => f (a -> b) -> f a -> f b  
Synonym for `<*>`.

**return** : (Applicative m) => a -> m a  
Inject a value into the monadic type.

**join** : (Action m) => m (m a) -> m a  
Collapses nested actions into a single action.

**identity** : a -> a  
The identity function.

**guard** : (ActionFail m) => Bool -> m ()

**foldl** : (b -> a -> b) -> b -> [a] -> b  
foldl f i xs performs a left fold over xs using f with the starting value i.  
Examples---

```
>>> foldl (+) 0 [1,2,3]
6
>>> foldl (^) 10 [2,3]
1000000
```

**find** : (a -> Bool) -> [a] -> Optional a  
find p xs finds the first element of xs where the predicate p holds.

**length** : [a] -> Int  
Calculate the length of a list.

**any** : (a -> Bool) -> [a] -> Bool  
any p xs is True if p holds for at least one element of xs.

**all** : (a -> Bool) -> [a] -> Bool  
all p xs is True if p holds for every element of xs.

**or** : [Bool] -> Bool  
or bs is True if at least one element of bs is True.

**and** : [Bool] -> Bool  
and bs is True if every element of bs is True.

**elem** : (Eq a) => a -> [a] -> Bool  
elem x xs is True if x is an element of the list xs.

**notElem** : (Eq a) => a -> [a] -> Bool  
Negation of elem.

**(<\$>)** : (Functor f) => (a -> b) -> f a -> f b  
Synonym for fmap.

**optional** : b -> (a -> b) -> Optional a -> b

The `optional` function takes a default value, a function, and a `Optional` value. If the `Optional` value is `None`, the function returns the default value. Otherwise, it applies the function to the value inside the `Some` and returns the result.

Examples---

Basic usage:

```
>>> optional False odd (Some 3)
True
```

```
>>> optional False odd None
False
```

Read an `Int` from a string using `readOptional`. If we succeed, return twice the `Int`; that is, apply `(*2)` to it. If instead we fail to parse an `Int`, return 0 by default:

```
>>> import Text.Read ( readOptional )
>>> optional 0 (*2) (readOptional "5")
10
>>> optional 0 (*2) (readOptional "")
0
```

Apply `show` to a `Optional Int`. If we have `Some n`, we want to show the underlying `Int`, `n`. But if we have `None`, we return the empty string instead of (for example) `None`:

```
>>> optional "" show (Some 5)
"5"
>>> optional "" show None
""
```

**either** : (a -> c) -> (b -> c) -> Either a b -> c

Case analysis for the `Either` type. If the value is `Left a`, apply the first function to `a`; if it is `Right b`, apply the second function to `b`.

Examples---

We create two values of type `Either String Int`, one using the `Left` constructor and another using the `Right` constructor. Then we apply either the `length` function (if we have a `String`) or the times-two function (if we have an `Int`):

```
>>> let s = Left "foo" :: Either String Int
>>> let n = Right 3 :: Either String Int
>>> either length (*2) s
3
>>> either length (*2) n
6
```

**concat** : [[a]] -> [a]**(++)** : [a] -> [a] -> [a]

Concatenate two lists.

**flip** : (a -> b -> c) -> b -> a -> c

Flip the order of the arguments of a two argument function.

**reverse** : [a] -> [a]

Reverse a list.

**mapA** : (Applicative m) => (a -> m b) -> [a] -> m [b]

Apply an applicative function to each element of a list.

**forA** : (Applicative m) => [a] -> (a -> m b) -> m [b]

forA is mapA with its arguments flipped.

**sequence** : (Applicative m) => [m a] -> m [a]

Perform a list of actions in sequence and collect the results.

**(=<<)** : (Action m) => (a -> m b) -> m a -> m b

=<< is >>= with its arguments flipped.

**concatMap** : (a -> [b]) -> [a] -> [b]

Map a function over each element of a list, and concatenate all the results.

**replicate** : Int -> a -> [a]

replicate i x is the list [x, x, x, ..., x] with i copies of x.

**take** : Int -> [a] -> [a]

Take the first n elements of a list.

**drop** : Int -> [a] -> [a]

Drop the first n elements of a list.

**splitAt** : Int -> [a] -> ([a], [a])

Split a list at a given index.

**takeWhile** : (a -> Bool) -> [a] -> [a]

Take elements from a list while the predicate holds.

**dropWhile** : (a -> Bool) -> [a] -> [a]

Drop elements from a list while the predicate holds.

**span** : (a -> Bool) -> [a] -> ([a], [a])

span p xs is equivalent to (takeWhile p xs, dropWhile p xs).

**break** : (a -> Bool) -> [a] -> ([a], [a])

Break a list into two, just before the first element where the predicate holds. break p xs is equivalent to span (not . p) xs.

**lookup** : (Eq a) => a -> [(a, b)] -> Optional b

Look up the first element with a matching key.

**zip** : [a] -> [b] -> [(a, b)]

zip takes two lists and returns a list of corresponding pairs. If one list is shorter, the excess elements of the longer list are discarded.

**zip3** : [a] -> [b] -> [c] -> [(a, b, c)]

zip3 takes three lists and returns a list of triples, analogous to zip.

**zipWith** : (a -> b -> c) -> [a] -> [b] -> [c]

zipWith generalises zip by combining elements using the function, instead of making pairs. If one list is shorter, the excess elements of the longer list are discarded.

**zipWith3** : (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]

zipWith3 generalises zip3 by combining elements using the function, instead of making pairs.

**unzip** : [(a, b)] -> ([a], [b])

Turn a list of pairs into a pair of lists.

**unzip3** :  $[(a, b, c)] \rightarrow ([a], [b], [c])$

Turn a list of triples into a triple of lists.

**traceRaw** : Text  $\rightarrow$  a  $\rightarrow$  a

traceRaw msg a prints msg and returns a, for debugging purposes.

**trace** : (Show b)  $\Rightarrow$  b  $\rightarrow$  a  $\rightarrow$  a

trace b a prints b and returns a, for debugging purposes.

**traceId** : (Show b)  $\Rightarrow$  b  $\rightarrow$  b

traceId a prints a and returns a, for debugging purposes.

**debug** : (Show b, Applicative m)  $\Rightarrow$  b  $\rightarrow$  m()

debug x prints x for debugging purposes.

**fst** : (a, b)  $\rightarrow$  a

Return the first element of a pair.

**snd** : (a, b)  $\rightarrow$  b

Return the second element of a pair.

**truncate** : Decimal  $\rightarrow$  Int

truncate x rounds x toward zero.

**intToDecimal** : Int  $\rightarrow$  Decimal

Convert an Int to a Decimal.

**roundBankers** : Int  $\rightarrow$  Decimal  $\rightarrow$  Decimal

Bankers' Rounding: roundBankers dp x rounds x to dp decimal places, where a .5 is rounded to the nearest even digit.

**roundCommercial** : Int  $\rightarrow$  Decimal  $\rightarrow$  Decimal

Commercial Rounding: roundCommercial dp x rounds x to dp decimal places, where a .5 is rounded away from zero.

**round** : Decimal  $\rightarrow$  Int

Round to nearest integer, where a .5 is rounded away from zero.

**floor** : Decimal  $\rightarrow$  Int

Round down to nearest integer.

**ceiling** : Decimal  $\rightarrow$  Int

Round up to nearest integer.

**null** : [a]  $\rightarrow$  Bool

null xs is true if xs is the empty list.

**filter** : (a  $\rightarrow$  Bool)  $\rightarrow$  [a]  $\rightarrow$  [a]

Keep only the elements where the predicate holds.

**sum** : (Additive a)  $\Rightarrow$  [a]  $\rightarrow$  a

Calculate the sum over all elements

**product** : (Multiplicative a)  $\Rightarrow$  [a]  $\rightarrow$  a

Calculate the product over all elements

**create** : (Template c)  $\Rightarrow$  c  $\rightarrow$  Update (ContractId c)

Create a contract based on a template.

**exercise** : ContractId c  $\rightarrow$  e  $\rightarrow$  Update r

Exercise a contract choice.

**fetch** : (Template c) => ContractId c -> Update c  
Fetch the contract data associated with the given contract id.

**archive** : (Template c) => ContractId c -> Update ()  
Archive the contract.

**stakeholder** : (Template c) => c -> [Party]  
The stakeholders of a contract, i.e. its signatories and observers.

## Module Control.Exception.Base

### Functions

**recSelError** : Text -> r

**absentSumFieldError** : a

**untangle** : Text -> Text -> Text

**append** : Text -> Text -> Text

**break** : (a -> Bool) -> [a] -> ([a], [a])

## Module DA.Action

### Action

### Functions

**when** : (Applicative f) => Bool -> f () -> f ()  
Conditional execution of Action expressions. For example,

```
when final (archive contractId)
```

will archive the contract `contractId` if the Boolean value `final` is True, and otherwise do nothing.

This function has short-circuiting semantics, i.e., when both arguments are present and the first argument evaluates to `False`, the second argument is not evaluated at all.

**unless** : (Applicative f) => Bool -> f () -> f ()

The reverse of `when`.

This function has short-circuiting semantics, i.e., when both arguments are present and the first argument evaluates to `False`, the second argument is not evaluated at all.

**foldrA** : (Action m) => (a -> b -> m b) -> b -> [a] -> m b

The `foldrA` is analogous to `foldr`, except that its result is encapsulated in an action. Note that `foldrA` works from right-to-left over the list arguments.

**foldr1A** : (Action m) => (a -> a -> m a) -> [a] -> m a

`foldr1A` is like `foldrA` but raises an error when presented with an empty list argument.

**foldlA** : (Action m) => (b -> a -> m b) -> b -> [a] -> m b

`foldlA` is analogous to `foldl`, except that its result is encapsulated in an action. Note that `foldlA` works from left-to-right over the list arguments.

**foldl1A** : (Action m) => (a -> a -> m a) -> [a] -> m a

The `foldl1A` is like `foldlA` but raises an errors when presented with an empty list argument.

**replicateA** : (Applicative m) => Int -> m a -> m [a]

`replicateA n act` performs the action `n` times, gathering the results.

## Module DA.Assert

### Assert

#### Functions

**(==)** : (CanAbort m, Show a, Eq a) => a -> a -> m ()

Check two values for equivalence and fail with a message if they are not.

**assertAfterMsg** : (CanAbort m, HasTime m) => Text -> Time -> m ()

Check whether the given time is in the future, and otherwise abort with a message.

**assertBeforeMsg** : (CanAbort m, HasTime m) => Text -> Time -> m ()

Check whether the given time is in the past, and otherwise abort with a message.

## Module DA.Bifunctor

### Typeclasses

**class Bifunctor p where** A bifunctor is a type constructor that takes two type arguments and is a functor in /both/ arguments. That is, unlike with ‘Functor’, a type constructor such as ‘Either’ does not need to be partially applied for a ‘Bifunctor’ instance, and the methods in this class permit mapping functions over the ‘Left’ value or the ‘Right’ value, or both at the same time. Intuitively it is a bifunctor where both the first and second arguments are covariant.

You can define a ‘Bifunctor’ by either defining ‘bimap’ or by defining both ‘first’ and ‘second’.

If you supply ‘bimap’, you should ensure that:

@‘bimap’ ‘id’ ‘id’ ‘id’@

If you supply ‘first’ and ‘second’, ensure:

@ ‘first’ ‘id’ ‘id’ ‘second’ ‘id’ ‘id’ @

If you supply both, you should also ensure:

@‘bimap’ f g ‘first’ f ‘second’ g@

These ensure by parametricity:

@ ‘bimap’ (f ‘:’ g) (h ‘:’ i) ‘bimap’ f h ‘:’ ‘bimap’ g i ‘first’ (f ‘:’ g) ‘first’ f ‘:’ ‘first’ g ‘second’ (f ‘:’ g) ‘second’ f ‘:’ ‘second’ g @

@since 4.8.0.0

**bimap** : (a -> b) -> (c -> d) -> p a c -> p b d

Map over both arguments at the same time.

@‘bimap’ f g ‘first’ f ‘second’ g@

**==== Examples**

>>> bimap toUpper (+1) ('j', 3) >>> ('J', 4)

>>> bimap toUpper (+1) (Left 'j') >>> Left 'J'

>>> bimap toUpper (+1) (Right 3) >>> Right 4

```
first : (a -> b) -> p a c -> p b c
Map covariantly over the first argument.
@'first' f 'bimap' f 'id'@
===== Examples
>>> first toUpper ('j', 3) >>> ('J',3)
>>> first toUpper (Left 'j') >>> Left 'J'
second : (b -> c) -> p a b -> p a c
Map covariantly over the second argument.
@'second' 'bimap' 'id'@
===== Examples
>>> second (+1) ('j', 3) >>> ('j',4)
>>> second (+1) (Right 3) >>> Right 4
```

## Module DA.Date

### Data Types

#### data DayOfWeek

The DayOfWeek type represents one the seven days of the week.

**Monday**

**Tuesday**

**Wednesday**

**Thursday**

**Friday**

**Saturday**

**Sunday**

#### data Month

The Month type represents a month in the Gregorian calendar.

**Jan**

**Feb**

**Mar**

**Apr**

**May**

**Jun**

**Jul**

**Aug**

**Sep**

**Oct**

**Nov**

**Dec****Functions****addDays** : Date → Int → Date

Adjusts a date with given number of days.

**subDate** : Date → Date → Int

Returns number of days between two given dates.

**dayOfWeek** : Date → DayOfWeek

Returns the day of week for given date

**fromGregorian** : (Int, Month, Int) → Date

Constructs a Date from the triplet (year, month, days).

**toGregorian** : Date → (Int, Month, Int)

Turn Date value into (year, month, day) triple, according to Gregorian calendar.

**date** : Int → Month → Int → Date

date (y, m, d) turns given year y, month m, and day d into a Date value.

**isLeapYear** : Int → Bool

Calculate whether the given year is a leap year.

**fromMonth** : Month → Int

Get the number corresponding to given month. For example, Jan corresponds to 1, Feb corresponds to 2, and so on.

**monthDayCount** : Int → Month → Int

Get number of days in month on given year, according to Gregorian calendar. This does not take historical calendar changes into account (for example, the move from Julian to Gregorian calendar), but does count leap years.

**datetime** : Int → Month → Int → Int → Int → Time

Constructs an instant using year, month, day, hours, minutes, seconds.

**toDateUTC** : Time → Date

Extracts UTC Date from UTC Time. This function will truncate Time to Date, but in many cases it will not return the date you really want. The reason for this is that usually the source of Time would be `getTime`, and `getTime` returns UTC, and most likely the date you want is something local to a location or an exchange. Consequently the date retrieved this way would be yesterday if retrieved when the market opens in say Singapore.

**passToDate** : Date → Scenario Time

Pass simulated scenario to given date.

**Module DA.Either**

The Either type represents values with two possibilities.

It is sometimes used to represent a value which is either correct or an error; by convention the `Left` constructor is used to hold an error value and the `Right` constructor is used to hold a correct value (mnemonic: right also means correct).

## Functions

**lefts** : [Either a b] -> [a]

Extracts all the Left elements.

**rights** : [Either a b] -> [b]

Extracts all the Right elements.

**partitionEithers** : [Either a b] -> ([a], [b])

Partitions a list of Either into two lists, the Left and Right elements respectively. Order is maintained.

**isLeft** : Either a b -> Bool

Return True if the given value is a Left-value, False otherwise.

**isRight** : Either a b -> Bool

Return True if the given value is a Right-value, False otherwise.

**fromLeft** : a -> Either a b -> a

Return the contents of a Left-value or a default value otherwise.

**fromRight** : b -> Either a b -> b

Return the contents of a Right-value or a default value otherwise.

**optionalToEither** : a -> Optional b -> Either a b

Convert a Optional value to an Either value, using the supplied parameter as the Left value if the Optional is None.

**eitherToOptional** : Either a b -> Optional b

Convert an Either value to a Optional, dropping any value in Left.

**maybeToEither** : a -> Optional b -> Either a b

**eitherToMaybe** : Either a b -> Optional b

## Module DA.Foldable

Class of data structures that can be folded to a summary value. You typically would want to import this module qualified to avoid clashes with functions defined in Prelude. Ie: `import DA.Foldable` qualified as F`

## Typeclasses

**class Foldable t where** Class of data structures that can be folded to a summary value.

**fold** : (Monoid m) => t m -> m

Combine the elements of a structure using a monoid.

**foldMap** : (Monoid m) => (a -> m) -> t a -> m

Combine the elements of a structure using a monoid.

**foldr** : (a -> b -> b) -> b -> t a -> b

Right-associative fold of a structure.

**foldl** : (b -> a -> b) -> b -> t a -> b

Left-associative fold of a structure.

**foldr1** : (a -> a -> a) -> t a -> a

A variant of foldr that has no base case, and thus should only be applied to non-empty structures.

**foldl1** : (a -> a -> a) -> t a -> a

A variant of foldl that has no base case, and thus should only be applied to non-empty structures.

**toList** : t a -> [a]

List of elements of a structure, from left to right.

**null** : t a -> Bool

Test whether the structure is empty. The default implementation is optimized for structures that are similar to cons-lists, because there is no general way to do better.

**length** : t a -> Int

Returns the size/length of a finite structure as an Int. The default implementation is optimized for structures that are similar to cons-lists, because there is no general way to do better.

**elem** : (Eq a) => a -> t a -> Bool

Does the element occur in the structure?

**sum** : (Additive a) => t a -> a

The sum function computes the sum of the numbers of a structure.

**product** : (Multiplicative a) => t a -> a

The product function computes the product of the numbers of a structure.

**minimum** : (Ord a) => t a -> a

The least element of a non-empty structure.

**maximum** : (Ord a) => t a -> a

The largest element of a non-empty structure.

## Functions

**concat** : (Foldable t) => t [a] -> [a]

The concatenation of all the elements of a container of lists.

**and** : (Foldable t) => t Bool -> Bool

and returns the conjunction of a container of Booleans. For the result to be True, the container must be finite; False, however, results from a False value finitely far from the left end.

**or** : (Foldable t) => t Bool -> Bool

or returns the disjunction of a container of Booleans. For the result to be False, the container must be finite; True, however, results from a True value finitely far from the left end.

**any** : (Foldable t) => (a -> Bool) -> t a -> Bool

Determines whether any element of the structure satisfies the predicate.

**all** : (Foldable t) => (a -> Bool) -> t a -> Bool

Determines whether all elements of the structure satisfy the predicate.

## Module DA.Functor

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

## Functions

**(\$>)** : (Functor f) => f a -> b -> f b

Replace all locations in the input (on the left) with the given value (on the right).

**(<&>)** : (Functor f) => f a -> (a -> b) -> f b

Map a function over a functor. Given a value as and a function f, as **<&>** f is f **<\$>** as. That is, **<&>** is like **<\$>** but the arguments are in reverse order.

**void** : (Functor f) => f a -> f ()

Replace all the locations in the input with () .

## Module DA.Internal.Compatible

Our Prelude, extending WiredIn with things that don't need special treatment.

## Data Types

**type Datetime** = Time

**type Integer** = Int

**type List a** = [a]

**type Maybe** = Optional

**type Monad** = Action

**type MonadFail** = ActionFail

**type Num** = Number

**type Tuple a b** = (a, b)

**type Tuple3 a b c** = (a, b, c)

## Functions

**tuple** : a -> b -> (a, b)

**tuple3** : a -> b -> c -> (a, b, c)

**nil** : [a]

**cons** : a -> [a] -> [a]

**does** : Party -> Update a -> Update a

**toText** : (Show a) => a -> Text

**toInteger** : Decimal -> Int

**mapU** : (Applicative m) => (a -> m b) -> [a] -> m [b]

**forU** : (Applicative m) => [a] -> (a -> m b) -> m [b]

**mapM** : (Applicative m) => (a -> m b) -> [a] -> m [b]

---

**forM** : (Applicative m) => [a] -> (a -> m b) -> m [b]  
**commits** : Party -> Update a -> Scenario a  
**fails** : Party -> Update a -> Scenario ()  
**test** : Scenario a -> Scenario a  
**maybe** : b -> (a -> b) -> Maybe a -> b  
**id** : a -> a

## Module DA.Internal.RebindableSyntax

Automatically imported unqualified in every module.

## Module DA.Internal.Record

### Typeclasses

```
class HasField x r a where
    getField : r -> a
    setField : a -> r -> r
```

### Functions

```
getFieldPrim : rec -> fld
setFieldPrim : fld -> rec -> rec
```

## Module DA.List

### List

### Functions

**sort** : (Ord a) => [a] -> [a]  
The `sort` function implements a stable sorting algorithm. It is a special case of `sortBy`, which allows the programmer to supply their own comparison function.  
Elements are arranged from lowest to highest, keeping duplicates in the order they appeared in the input (a stable sort).

**sortBy** : (a -> a -> Ordering) -> [a] -> [a]  
The `sortBy` function is the non-overloaded version of `sort`.

**sortOn** : (Ord k) => (a -> k) -> [a] -> [a]  
Sort a list by comparing the results of a key function applied to each element. `sortOn f` is equivalent to `sortBy (comparing f)`, but has the performance advantage of only evaluating

f once for each element in the input list. This is sometimes called the decorate-sort-undecorate paradigm.

Elements are arranged from lowest to highest, keeping duplicates in the order they appeared in the input.

**mergeBy** : (a -> a -> Ordering) -> [a] -> [a] -> [a]

Merge two sorted lists into a single, sorted whole, allowing the programmer to specify the comparison function.

**combinePairs** : (a -> a -> a) -> [a] -> [a]

Combine elements pairwise by means of a programmer supplied function from two list inputs into a single list.

**foldBalanced1** : (a -> a -> a) -> [a] -> a

Fold a non-empty list in a balanced way. Balanced means that each element has approximately the same depth in the operator tree. Approximately the same depth means that the difference between maximum and minimum depth is at most 1. The accumulation operation must be associative and commutative in order to get the same result as foldl1 or foldr1.

**group** : (Eq a) => [a] -> [[a]]

The ‘group’ function groups equal elements into sublists such that the concatenation of the result is equal to the argument.

**groupBy** : (a -> a -> Bool) -> [a] -> [[a]]

The ‘groupBy’ function is the non-overloaded version of ‘group’.

**groupOn** : (Eq k) => (a -> k) -> [a] -> [[a]]

Similar to ‘group’, except that the equality is done on an extracted value.

**dedup** : (Ord a) => [a] -> [a]

dedup 1 removes duplicate elements from a list. In particular, it keeps only the first occurrence of each element. It is a special case of dedupBy, which allows the programmer to supply their own equality test. dedup is called nub in Haskell.

**dedupBy** : (a -> a -> Ordering) -> [a] -> [a]

A version of dedup with a custom predicate.

**dedupOn** : (Ord k) => (a -> k) -> [a] -> [a]

A version of dedup where deduplication is done after applying function. Example use: dedupOn (.employeeNo) employees

**dedupSort** : (Ord a) => [a] -> [a]

The dedupSort function sorts and removes duplicate elements from a list. In particular, it keeps only the first occurrence of each element.

**dedupSortBy** : (a -> a -> Ordering) -> [a] -> [a]

A version of dedupSort with a custom predicate.

**unique** : (Ord a) => [a] -> Bool

Returns True if and only if there are no duplicate elements in the given list.

**uniqueBy** : (a -> a -> Ordering) -> [a] -> Bool

A version of unique with a custom predicate.

**uniqueOn** : (Ord k) => (a -> k) -> [a] -> Bool

Returns True if and only if there are no duplicate elements in the given list after applying function. Example use: assert \$ uniqueOn (.employeeNo) employees

**replace** : (Eq a) => [a] -> [a] -> [a] -> [a]

Given a list and a replacement list, replaces each occurrence of the search list with the replacement list in the operation list.

**dropPrefix** : (Eq a) => [a] -> [a] -> [a]

Drops the given prefix from a list. It returns the original sequence if the sequence doesn't start with the given prefix.

**dropSuffix** : (Eq a) => [a] -> [a] -> [a]

Drops the given suffix from a list. It returns the original sequence if the sequence doesn't end with the given suffix.

**stripPrefix** : (Eq a) => [a] -> [a] -> Optional [a]

The `stripPrefix` function drops the given prefix from a list. It returns `None` if the list did not start with the prefix given, or `Some` the list after the prefix, if it does.

**stripSuffix** : (Eq a) => [a] -> [a] -> Optional [a]

Return the prefix of the second list if its suffix matches the entire first list.

**isPrefixOf** : (Eq a) => [a] -> [a] -> Bool

The `isPrefixOf` function takes two lists and returns `True` if and only if the first is a prefix of the second.

**isSuffixOf** : (Eq a) => [a] -> [a] -> Bool

The `isSuffixOf` function takes two lists and returns `True` if and only if the first list is a suffix of the second.

**isInfixOf** : (Eq a) => [a] -> [a] -> Bool

The `isInfixOf` function takes two lists and returns `True` if and only if the first list is contained anywhere within the second.

**mapAccumL** : (acc -> x -> (acc, y)) -> acc -> [x] -> (acc, [y])

The `mapAccumL` function combines the behaviours of `map` and `foldl`; it applies a function to each element of a list, passing an accumulating parameter from left to right, and returning a final value of this accumulator together with the new list.

**inits** : [a] -> [[a]]

The `inits` function returns all initial segments of the argument, shortest first.

**intersperse** : a -> [a] -> [a]

The `intersperse` function takes an element and a list and intersperses that element between the elements of the list.

**intercalate** : [a] -> [[a]] -> [a]

`intercalate` inserts the list `xs` in between the lists in `xss` and concatenates the result.

**tails** : [a] -> [[a]]

The `tails` function returns all final segments of the argument, longest first.

**dropWhileEnd** : (a -> Bool) -> [a] -> [a]

A version of `dropWhile` operating from the end.

**takeWhileEnd** : (a -> Bool) -> [a] -> [a]

A version of `takeWhile` operating from the end.

**transpose** : [[a]] -> [[a]]

The `transpose` function transposes the rows and columns of its argument.

**breakEnd** : (a -> Bool) -> [a] -> ([a], [a])

Break, but from the end.

**breakOn** : (Eq a) => [a] -> [a] -> ([a], [a])

Find the first instance of `needle` in `haystack`. The first element of the returned tuple is the prefix of `haystack` before `needle` is matched. The second is the remainder of `haystack`, starting with the match. If you want the remainder without the match, use `stripInfix`.

**breakOnEnd** : (Eq a) => [a] -> [a] -> ([a], [a])

Similar to `breakOn`, but searches from the end of the string.

The first element of the returned tuple is the prefix of `haystack` up to and including the last match of `needle`. The second is the remainder of `haystack`, following the match.

**linesBy** : (a -> Bool) -> [a] -> [[a]]

A variant of `lines` with a custom test. In particular, if there is a trailing separator it will be discarded.

**wordsBy** : (a -> Bool) -> [a] -> [[a]]

A variant of `words` with a custom test. In particular, adjacent separators are discarded, as are leading or trailing separators.

**head** : [a] -> a

Extract the first element of a list, which must be non-empty.

**tail** : [a] -> [a]

Extract the elements after the head of a list, which must be non-empty.

**last** : [a] -> a

Extract the last element of a list, which must be finite and non-empty.

**init** : [a] -> [a]

Return all the elements of a list except the last one. The list must be non-empty.

**foldl1** : (a -> a -> a) -> [a] -> a

Left associative fold of a list that must be non-empty.

**foldr1** : (a -> a -> a) -> [a] -> a

Right associative fold of a list that must be non-empty.

**repeatedly** : ([a] -> (b, [a])) -> [a] -> [b]

Apply some operation repeatedly, producing an element of output and the remainder of the list.

**(!!)** : [a] -> Int -> a

List index (subscript) operator, starting from 0. For example, `xs !! 2` returns the third element in `xs`. Raises an error if the index is not suitable for the given list. The function has complexity  $O(n)$  where  $n$  is the index given, unlike in languages such as Java where array indexing is  $O(1)$ .

**elemIndex** : (Eq a) => a -> [a] -> Optional Int

Find index of element in given list. Will return `None` if not found.

**findIndex** : (a -> Bool) -> [a] -> Optional Int

Find index, given predicate, of first matching element. Will return `None` if not found.

**findIndex\_** : Int -> (a -> Bool) -> [a] -> Optional Int

Module DA.List.Total

## Functions

```
head : (ActionFail m) => [a] -> m a
tail : (ActionFail m) => [a] -> m [a]
last : (ActionFail m) => [a] -> m a
init : (ActionFail m) => [a] -> m [a]
(!!) : (ActionFail m) => [a] -> Int -> m a
foldl1 : (ActionFail m) => (a -> a -> a) -> [a] -> m a
foldr1 : (ActionFail m) => (a -> a -> a) -> [a] -> m a
foldBalanced1 : (ActionFail m) => (a -> a -> a) -> [a] -> m a
```

## Module DA.Logic

Logic - Propositional calculus.

### Data Types

**data Formula t**

A `Formula t` is a formula in propositional calculus with propositions of type `t`.

**Proposition t**

`Proposition p` is the formula `p`

**Negation Formula t**

For a formula `f`, `Negation f` is `f`

**Conjunction [Formula t]**

For formulas `f1, , fn`, `Conjunction [f1, , fn]` is `f1 & fn`

**Disjunction [Formula t]**

For formulas `f1, , fn`, `Disjunction [f1, , fn]` is `f1 | fn`

### Functions

**(&&&)** : `Formula t -> Formula t -> Formula t`

`&&&` is the operation of the boolean algebra of formulas, to be read as and

**(|||)** : `Formula t -> Formula t -> Formula t`

`|||` is the operation of the boolean algebra of formulas, to be read as or

**true** : `Formula t`

`true` is the 1 element of the boolean algebra of formulas, represented as an empty conjunction.

**false** : `Formula t`

`false` is the 0 element of the boolean algebra of formulas, represented as an empty disjunction.

**neg** : Formula t -> Formula t

neg is the (negation) operation of the boolean algebra of formulas.

**conj** : [Formula t] -> Formula t

conj is a list version of &&&, enabled by the associativity of .

**disj** : [Formula t] -> Formula t

disj is a list version of |||, enabled by the associativity of .

**fromBool** : Bool -> Formula t

fromBool converts True to true and False to false.

**toNNF** : Formula t -> Formula t

toNNF transforms a formula to negation normal form (see [https://en.wikipedia.org/wiki/Negation\\_normal\\_form](https://en.wikipedia.org/wiki/Negation_normal_form)).

**toDNF** : Formula t -> Formula t

toDNF turns a formula into disjunctive normal form. (see [https://en.wikipedia.org/wiki/Disjunctive\\_normal\\_form](https://en.wikipedia.org/wiki/Disjunctive_normal_form)).

**traverse** : (Applicative f) => (t -> f s) -> Formula t -> f (Formula s)

An implementation of traverse in the usual sense.

**zipFormulas** : Formula t -> Formula s -> Formula (t, s)

zipFormulas takes two formulas of same shape, meaning only propositions are different and zips them up.

**substitute** : (t -> Optional Bool) -> Formula t -> Formula t

substitute takes a truth assignment and substitutes True or False into the respective places in a formula.

**reduce** : Formula t -> Formula t

reduce reduces a formula as far as possible by:

1. Removing any occurrences of true and false;
2. Removing directly nested Conjunctions and Disjunctions;
3. Going to negation normal form.

**isBool** : Formula t -> Optional Bool

isBool attempts to convert a formula to a bool. It satisfies isBool true == Right True and isBool false == Right False. Otherwise, it returns Left x, where x is the input.

**interpret** : (t -> Optional Bool) -> Formula t -> Either (Formula t) Bool

interpret is a version of toBool that first substitutes using a truth function and then reduces as far as possible.

**substituteA** : (Applicative f) => (t -> f (Optional Bool)) -> Formula t -> f (Formula t)

substituteA is a version of substitute that allows for truth values to be obtained from an action.

**interpretA** : (Applicative f) => (t -> f (Optional Bool)) -> Formula t -> f (Either (Formula t) Bool)

interpretA is a version of interpret that allows for truth values to be obtained from an action.

## Module DA.Map

Map - A map is an associative array data type composed of a collection of key/value pairs such that, each possible key appears at most once in the collection.

## Data Types

**data Map k v**

The type of a Map from keys of type k to values of type v.

**Map\_internal [(k, v)]**

## Functions

**fromList : (Ord k) => [(k, a)] -> Map k a**

Create a map from a list of key/value pairs.

**fromListWith : (Ord k) => (a -> a -> a) -> [(k, a)] -> Map k a**

Create a map from a list of key/value pairs with a combining function. Examples:

```
fromListWith (<>) [(5, "a"), (5, "b"), (3, "b"), (3, "a"), (5, "c")] ==
  ↪fromList [(3, "ab"), (5, "cba")]
fromListWith (<>) [] == (empty : Map Int Text)
```

**fromListWithKey : (Ord k) => (k -> a -> a -> a) -> [(k, a)] -> Map k a**

Build a map from a list of key/value pairs with a combining function. Example:

```
let f key new_value old_value = (show key) ++ ":" ++ new_value ++ "|" ++
  ++ old_value
fromListWithKey f [(5, "a"), (5, "b"), (3, "b"), (3, "a"), (5, "c")] ==
  ↪fromList [(3, "3:a|b"), (5, "5:c|5:b|a")]
fromListWithKey f [] == empty
```

**toList : Map k v -> [(k, v)]**

Convert the map to a list of key/value pairs where the keys are in ascending order.

**empty : Map k v**

The empty map.

**size : Map k v -> Int**

Number of elements in the map.

**null : Map k v -> Bool**

Is the map empty?

**lookup : (Eq k, Ord k) => k -> Map k v -> Optional v**

Lookup the value at a key in the map.

**member : (Eq k, Ord k) => k -> Map k v -> Bool**

Is the key a member of the map?

**filter : (Eq k) => (k -> v -> Bool) -> Map k v -> Map k v**

Filter all values that satisfy some predicate.

**delete : (Eq k) => k -> Map k v -> Map k v**

Delete a key and its value from the map. When the key is not a member of the map, the original map is returned.

**insert : (Ord k) => k -> v -> Map k v -> Map k v**

Insert a new key/value pair in the map. If the key is already present in the map, the associated value is replaced with the supplied value.

**merge** : (Ord k) => (k -> a -> Optional c) -> (k -> b -> Optional c) -> (k -> a -> b -> Optional c) -> Map k a -> Map k b -> Map k c  
Merge two maps.

**union** : (Ord k) => Map k a -> Map k a -> Map k a

The union of two maps, preferring the first map when equal keys are encountered.

## Module DA.Maybe.Total

### Functions

**fromJust** : (ActionFail m) => Optional a -> m a

**fromJustNote** : (ActionFail m) => Text -> Optional a -> m a

## Module DA.Monoid

### Data Types

#### data All

Boolean monoid under conjunction (&&)

##### All

Field	Type	Description
getAll	Bool	

#### data Any

Boolean Monoid under disjunction (||)

##### Any

Field	Type	Description
getAny	Bool	

#### data Endo a

The monoid of endomorphisms under composition.

##### Endo

Field	Type	Description
appEndo	a -> a	

## Module DA.Optional

The `Optional` type encapsulates an optional value. A value of type `Optional a` either contains a value of type `a` (represented as `Some a`), or it is empty (represented as `None`). Using `Optional` is a good way to deal with errors or exceptional cases without resorting to drastic measures such as `error`.

The `Optional` type is also an action. It is a simple kind of error action, where all errors are represented by `None`. A richer error action can be built using the `Either` type.

## Functions

**fromSome** : `Optional a -> a`

The `fromSome` function extracts the element out of a `Some` and throws an error if its argument is `None`.

**fromSomeNote** : `Text -> Optional a -> a`

**catOptionals** : `[Optional a] -> [a]`

The `catOptionals` function takes a list of `Optional`s and returns a list of all the `Some` values.

**listToOptional** : `[a] -> Optional a`

The `listToOptional` function returns `None` on an empty list or `Some a` where `a` is the first element of the list.

**optionalToList** : `Optional a -> [a]`

The `optionalToList` function returns an empty list when given `None` or a singleton list when not given `None`.

**fromOptional** : `a -> Optional a -> a`

The `fromOptional` function takes a default value and a `Optional` value. If the `Optional` is `None`, it returns the default values otherwise, it returns the value contained in the `Optional`.

**isSome** : `Optional a -> Bool`

The `isSome` function returns `True` iff its argument is of the form `Some _`.

**isNone** : `Optional a -> Bool`

The `isNone` function returns `True` iff its argument is `None`.

**mapOptional** : `(a -> Optional b) -> [a] -> [b]`

The `mapOptional` function is a version of `map` which can throw out elements. In particular, the functional argument returns something of type `Optional b`. If this is `None`, no element is added on to the result list. If it is `Some b`, then `b` is included in the result list.

**whenSome** : `(Applicative m) => Optional a -> (a -> m ()) -> m ()`

Perform some operation on `Some`, given the field inside the `Some`.

## Module DA.Optional.Total

## Functions

**fromSome** : `(ActionFail m) => Optional a -> m a`

**fromSomeNote** : `(ActionFail m) => Text -> Optional a -> m a`

## Module DA.Record

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

## Module DA.Set

Set - The Set e type represents a set of elements of type e. Most operations require that e be an instance of the Ord class.

## Data Types

**data Set a**

The type of a set.

**Set\_internal M.Map a ()**

## Functions

**empty : Set a**

The empty set.

**size : Set a -> Int**

The number of elements in the set.

**toList : Set a -> [a]**

Convert the set to a list of elements.

**fromList : (Ord a) -> [a] -> Set a**

Create a set from a list of elements.

**member : (Ord a) -> a -> Set a -> Bool**

Is the element in the set?

**null : Set a -> Bool**

Is this the empty set?

**insert : (Ord a) -> a -> Set a -> Set a**

Insert an element in a set. If the set already contains an element equal to the given value, it is replaced with the new value.

**filter : (Eq a) -> (a -> Bool) -> Set a -> Set a**

Filter all elements that satisfy the predicate.

**delete : (Eq a) -> a -> Set a -> Set a**

Delete an element from a set.

**singleton : (Ord a) -> a -> Set a**

Create a singleton set.

**union : (Ord a) -> Set a -> Set a -> Set a**

The union of two sets, preferring the first set when equal elements are encountered.

**intersection** : (Ord a) => Set a -> Set a -> Set a

The intersection of two sets. Elements of the result come from the first set.

**difference** : (Ord a) => Set a -> Set a -> Set a

Difference of two sets.

## Module DA.Text

Functions for working with Text.

### Functions

**explode** : Text -> [Text]

**implode** : [Text] -> Text

**isEmpty** : Text -> Bool

Test for emptiness.

**length** : Text -> Int

Compute the number of symbols in the text.

**trim** : Text -> Text

Remove spaces from either side of the given text.

**replace** : Text -> Text -> Text -> Text

Replace a subsequence everywhere it occurs. The first argument must not be empty.

**lines** : Text -> [Text]

Breaks a Text value up into a list of Text's at newline symbols. The resulting texts do not contain newline symbols.

**unlines** : [Text] -> Text

Joins lines, after appending a terminating newline to each.

**words** : Text -> [Text]

Breaks a 'Text' up into a list of words, delimited by symbols representing white space.

**unwords** : [Text] -> Text

Joins words using single space symbols.

**linesBy** : (Text -> Bool) -> Text -> [Text]

A variant of lines with a custom test. In particular, if there is a trailing separator it will be discarded.

**wordsBy** : (Text -> Bool) -> Text -> [Text]

A variant of words with a custom test. In particular, adjacent separators are discarded, as are leading or trailing separators.

**intercalate** : Text -> [Text] -> Text

intercalate inserts the text argument t in between the items in ts and concatenates the result.

**dropPrefix** : Text -> Text -> Text

dropPrefix drops the given prefix from the argument. It returns the original text if the text doesn't start with the given prefix.

**dropSuffix** : Text -> Text -> Text

Drops the given suffix from the argument. It returns the original text if the text doesn't end with the given suffix. Examples:

```
dropSuffix "!" "Hello World!" == "Hello World"  
dropSuffix "!!" "Hello World!!" == "Hello World!"  
dropSuffix "!" "Hello World." == "Hello World."
```

**stripSuffix** : Text -> Text -> Optional Text

Return the prefix of the second text if its suffix matches the entire first text. Examples:

```
stripSuffix "bar" "foobar" == Some "foo"  
stripSuffix "" "baz" == Some "baz"  
stripSuffix "foo" "quux" == None
```

**stripPrefix** : Text -> Text -> Optional Text

The `stripPrefix` function drops the given prefix from the argument text. It returns `None` if the text did not start with the prefix.

**isPrefixOf** : Text -> Text -> Bool

The `isPrefixOf` function takes two text arguments and returns `True` if and only if the first is a prefix of the second.

**isSuffixOf** : Text -> Text -> Bool

The `isSuffixOf` function takes two text arguments and returns `True` if and only if the first is a suffix of the second.

**isInfixOf** : Text -> Text -> Bool

The `isInfixOf` function takes two text arguments and returns `True` if and only if the first is contained, wholly and intact, anywhere within the second.

**takeWhile** : (Text -> Bool) -> Text -> Text

The function `takeWhile`, applied to a predicate `p` and a text, returns the longest prefix (possibly empty) of symbols that satisfy `p`.

**takeWhileEnd** : (Text -> Bool) -> Text -> Text

The function ‘`takeWhileEnd`’, applied to a predicate `p` and a ‘Text’, returns the longest suffix (possibly empty) of elements that satisfy `p`.

**dropWhile** : (Text -> Bool) -> Text -> Text

`dropWhile p t` returns the suffix remaining after `takeWhile p t`.

**dropWhileEnd** : (Text -> Bool) -> Text -> Text

`dropWhileEnd p t` returns the prefix remaining after dropping symbols that satisfy the predicate `p` from the end of `t`.

**splitOn** : Text -> Text -> [Text]

Break a text into pieces separated by the first text argument (which cannot be empty), consuming the delimiter.

**take** : Int -> Text -> Text

`take n t`, applied to a text `t`, returns the prefix of `t` of length `n`, or `t` itself if `n` is greater than the length of `t`.

**drop** : Int -> Text -> Text

`drop n t`, applied to a text `t`, returns the suffix of `t` after the first `n` characters, or the empty Text if `n` is greater than the length of `t`.

**substring** : Int → Int → Text → Text

Compute the sequence of symbols of length `l` in the argument `text` starting at `s`.

**isPred** : (Text → Bool) → Text → Bool

`isPred f t` returns `True` if `t` is not empty and `f` is `True` for all symbols in `t`.

**isSpace** : Text → Bool

`isSpace t` is `True` if `t` is not empty and consists only of spaces.

**isNewLine** : Text → Bool

`isSpace t` is `True` if `t` is not empty and consists only of newlines.

**isUpper** : Text → Bool

`isUpper t` is `True` if `t` is not empty and consists only of uppercase symbols.

**isLower** : Text → Bool

`isLower t` is `True` if `t` is not empty and consists only of lowercase symbols.

**isDigit** : Text → Bool

`isDigit t` is `True` if `t` is not empty and consists only of digit symbols.

**isAlpha** : Text → Bool

`isAlpha t` is `True` if `t` is not empty and consists only of alphabet symbols.

**isAlphaNum** : Text → Bool

`isAlphaNum t` is `True` if `t` is not empty and consists only of alphanumeric symbols.

**parseInt\_** : [Text] → Int → Optional Int**parseInt** : Text → Optional Int

Attempt to parse an `Int` value from a given `Text`.

**parsePositiveDecimal** : [Text] → Optional Decimal**parseDecimal** : Text → Optional Decimal

Attempt to parse a `Decimal` value from a given `Text`. To get `Some` value, the text must follow the regex `'-'?[0-9]+(:'[0-9]+)?`. In particular, the shorthands ".12" and "12." do not work. Leading and trailing zeros are fine, however spaces are not. Examples:

```
parseDecimal "3.14" == Some 3.14
parseDecimal "+12.0" == None
```

**sha256** : Text → Text

Computes the SHA256 of the UTF8 bytes of the `Text`, and returns it in its hex-encoded form.

The hex encoding uses lowercase letters.

This function will crash at runtime if you compile DAML to DAML-LF < 1.2.

## Module DA.Time

### Data Types

**data RelTime**

The `RelTime` type describes a time offset, i.e. relative time.

**RelTime**

Field	Type	Description
microseconds	Int	

## Functions

**time** : Date → Int → Int → Int → Time

time d h m s turns given UTC date d and the UTC time (given in hours, minutes, seconds) into a UTC timestamp (Time). Does not handle leap seconds.

**pass** : RelTime → Scenario Time

Pass simulated scenario time by argument

**addRelTime** : Time → RelTime → Time

Adjusts Time with given time offset.

**subTime** : Time → Time → RelTime

Returns time offset between two given instants.

**wholeDays** : RelTime → Int

Returns the number of whole days in a time offset. Fraction of time is rounded towards zero.

**days** : Int → RelTime

A number of days in relative time.

**hours** : Int → RelTime

A number of hours in relative time.

**minutes** : Int → RelTime

A number of minutes in relative time.

**seconds** : Int → RelTime

**convertRelTimeToMicroseconds** : RelTime → Int

Convert RelTime to microseconds Use higher level functions instead of the internal microseconds

**convertMicrosecondsToRelTime** : Int → RelTime

Convert microseconds to RelTime Use higher level functions instead of the internal microseconds

## Module DA.Traversable

Class of data structures that can be traversed from left to right, performing an action on each element. You typically would want to import this module qualified to avoid clashes with functions defined in Prelude. Ie.: `import DA.Traversable qualified as F`

## Typeclasses

**class (Functor t, Foldable t) => Traversable t where** Functors representing data structures that can be traversed from left to right.

**mapA** : (Applicative f) => (a → f b) → t a → f (t b)

Map each element of a structure to an action, evaluate these actions from left to right, and collect the results.

**sequence** : (Applicative f) => t (f a) -> f (t a)

Evaluate each action in the structure from left to right, and collect the results.

## Functions

**forA** : (Traversable t, Applicative f) => t a -> (a -> f b) -> f (t b)

`forA` is `mapA` with its arguments flipped.

## Module DA.Tuple

Tuple - Ubiquitous functions of tuples.

### Functions

**first** : (a -> a') -> (a, b) -> (a', b)

The pair obtained from a pair by application of a programmer supplied function to the argument pair's first field.

**second** : (b -> b') -> (a, b) -> (a, b')

The pair obtained from a pair by application of a programmer supplied function to the argument pair's second field.

**both** : (a -> b) -> (a, a) -> (b, b)

The pair obtained from a pair by application of a programmer supplied function to both the argument pair's first and second fields.

**swap** : (a, b) -> (b, a)

The pair obtained from a pair by permuting the order of the argument pair's first and second fields.

**dupe** : a -> (a, a)

Duplicate a single value into a pair.

> `dupe 12 == (12, 12)`

**fst3** : (a, b, c) -> a

Extract the 'fst' of a triple.

**snd3** : (a, b, c) -> b

Extract the 'snd' of a triple.

**thd3** : (a, b, c) -> c

Extract the final element of a triple.

**curry3** : ((a, b, c) -> d) -> a -> b -> c -> d

Converts an uncurried function to a curried function.

**uncurry3** : (a -> b -> c -> d) -> (a, b, c) -> d

Converts a curried function to a function on a triple.

## Module Data.String

## Functions

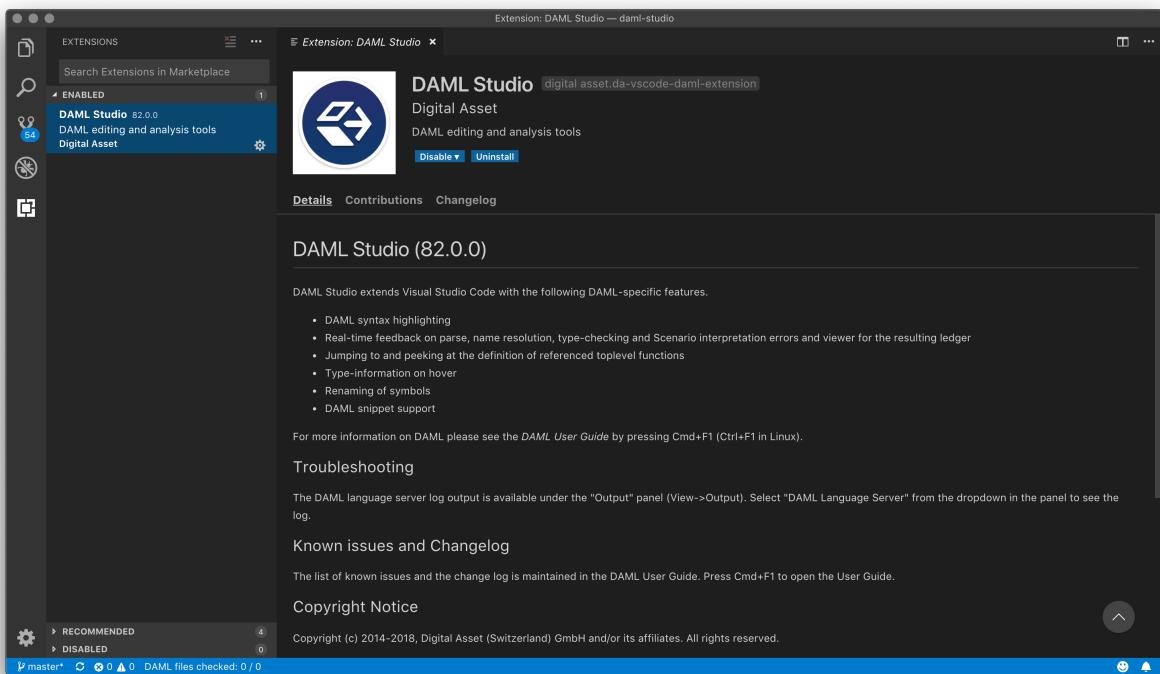
**fromString** : TextLit -> Text

## 2.3 DAML Studio

DAML Studio is an integrated development environment (IDE) for DAML. It is built on top of [Visual Studio Code](#) (VS Code), a cross-platform, open-source editor providing a [rich code editing experience](#). DAML-specific features are provided via the DAML Studio extension for Visual Studio Code.

### 2.3.1 Creating your first DAML file using DAML Studio

1. Start Visual Studio Code. To start it in the current project, use the `da studio` command. Alternatively you can simply start VS Code as you would normally start any application.
2. Check that the DAML Studio extension is installed by first clicking on the Extensions icon at the bottom of the VS Code sidebar, and then clicking on the DAML Studio extension that should be listed on the pane.



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

```
-- Copyright (c) 2019 Digital Asset (Switzerland) GmbH and/or its
-- affiliates. All rights reserved.
-- SPDX-License-Identifier: Apache-2.0

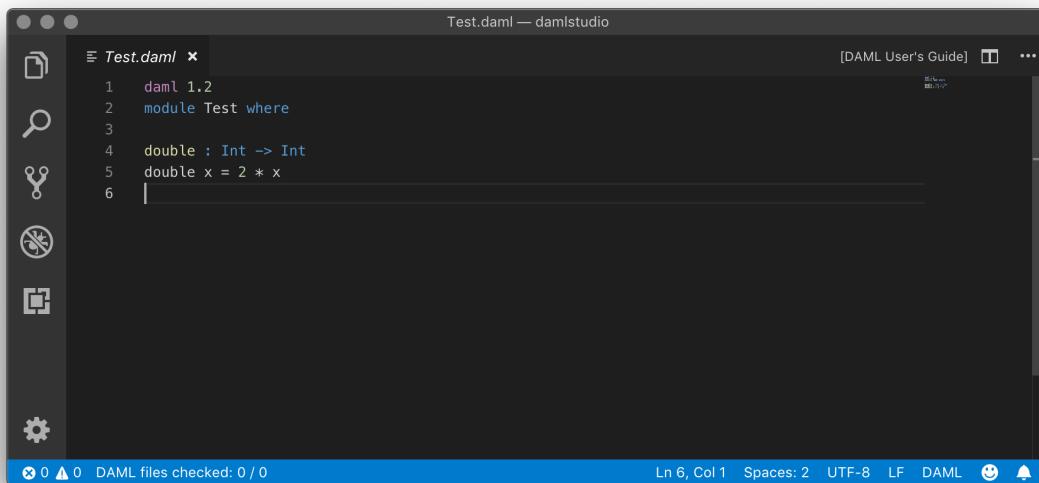
daml 1.2
module Test where
```

(continues on next page)

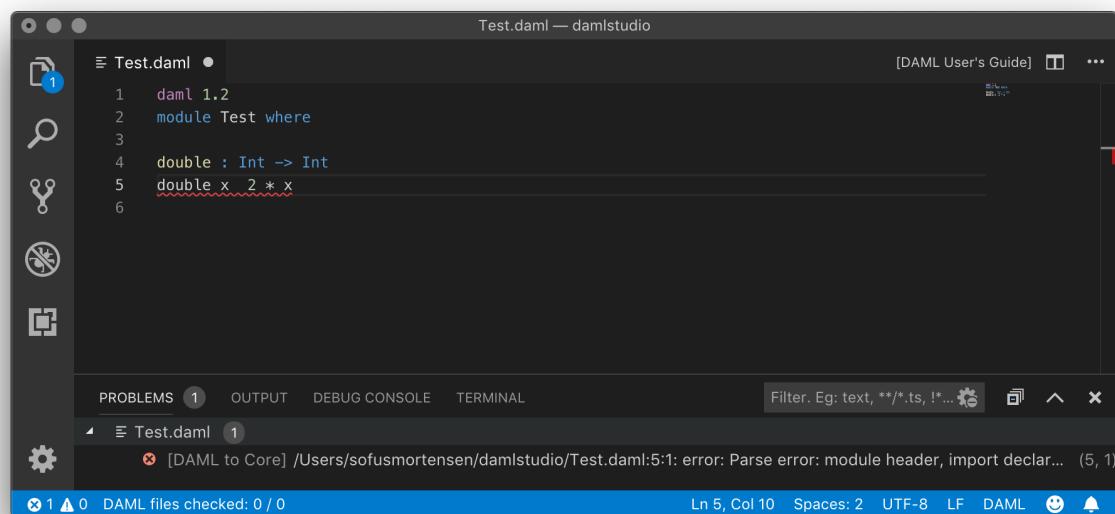
(continued from previous page)

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

Your screen should now look like the image below.



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



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

We recommend reviewing the [Visual Studio Code documentation](#) to learn more about how to use it. To learn more about DAML, see [DAML reference docs](#).

## 2.3.2 Supported features

Visual Studio Code provides many helpful features for editing DAML files and Digital Asset recommends reviewing [Visual Studio Code Basics](#) and [Visual Studio Code Keyboard Shortcuts for OS X](#). The DAML Studio extension for Visual Studio Code provides the following DAML-specific features.

### 2.3.2.1 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)
<a href="#">Go to Definition</a>	F12
<a href="#">Peek Definition</a>	F12
<a href="#">Rename Symbol</a>	F2
<a href="#">Go to Symbol in File</a>	O
<a href="#">Go to Symbol in Workspace</a>	T
<a href="#">Find all References</a>	F12
<a href="#">Problems Panel</a>	M

---

**Note:** You can also start a command by typing its name into the command palette (press P or F1). The command palette is also handy for looking up keyboard shortcuts.

---

**Note:**

[Rename Symbol](#), [Go to Symbol in File](#), [Go to Symbol in Workspace](#), and [Find all References](#) work on: choices, record fields, top-level definitions, let-bound variables, lambda arguments, and modules

[Go to Definition](#) and [Peek Definition](#) work on: top-level definitions, let-bound variables, lambda arguments, and modules

---

### 2.3.2.2 Hover tooltips

You can [hover](#) over most symbols in the code to display additional information such as its type.

### 2.3.2.3 Scenario results

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

The scenario results present a simplified view of a ledger, in the form of a transaction graph, after execution of the scenario. The transaction graph consists of transactions, each of which contain one or more updates to the ledger, that is creates and exercises. The transaction graph also records fetches of contracts.

For example a scenario for the `Iou` module looks as follows:

The screenshot shows the `Iou.daml` file open in the left pane. The right pane displays the results of running the scenario. The "Transactions" section shows two transactions: TX #0 and TX #1. TX #0 is the creation of an `Iou` contract by the bank, with arguments: issuer = 'Bank', owner = 'Alice', amount = 10, currency = "USD". TX #1 is the transfer of the `Iou` from Alice to Bob, with arguments: newOwner = 'Bob'. The "Active contracts" section shows the remaining contract #1:2.

```

Scenario: run — damlstudio
Iou.daml x ...
...
Scenario results
18 run = scenario do
19   bank <- getParty "Bank"
20   alice <- getParty "Alice"
21   bob <- getParty "Bob"
22   cid <- submit bank do
23     create Iou with
24       issuer = bank
25       owner = alice
26       amount = 10
27       currency = "USD"
28     submit alice do
29       exercise cid Transfer with
30         newOwner = bob
31
32
33

Transactions:
TX #0 1970-01-01T00:00:00Z (unknown source)
#0:0
  consumed by: #1:1
  referenced by #1:0, #1:1
  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 (unknown source)
#1:0
  fetch #0:0 (Iou:Iou)

#1:1
  known to (since): 'Bank' (#1), 'Alice' (#1)
  'Alice' exercises Transfer on #0:0 (Iou:Iou)
  with
    newOwner = 'Bob'
  children:
#1:2
  known to (since): 'Bank' (#1), 'Bob' (#1), 'Alice' (#1)
  create Iou:Iou
  with
    issuer = 'Bank'; owner = 'Bob'; amount = 10; currency = "USD"

Active contracts: #1:2
Return value: #1:2

```

Fig. 1: Scenario results (click to zoom)

Each transaction is the result of executing a step in the scenario. In the image below, the transaction #0 is the result of executing the first line of the scenario (line 20), where the `Iou` is created by the bank. The following information can be gathered from the transaction:

- The result of the first scenario transaction #0 was the creation of the `Iou` contract with the arguments bank, 10, and "USD".
- The created contract is referenced in transaction #1, step 0.
- The created contract was consumed in transaction #1, step 0.
- A new contract was created in transaction #1, step 1, and has been divulged to parties 'Alice', 'Bob', and 'Bank'.
- At the end of the scenario only the contract created in #1:1 remains.
- The return value from running the scenario is the contract identifier #1:2.
- And finally, the contract identifiers assigned in scenario execution correspond to the scenario step that created them (e.g. #1).

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

#### 2.3.2.4 DAML snippets

You can automatically complete a number of snippets when editing a DAML source file. By default, hitting `^Space` after typing a DAML keyword displays available snippets that you can insert.

To define your own workflow around DAML snippets, adjust your user settings in Visual Studio Code to include the following options:

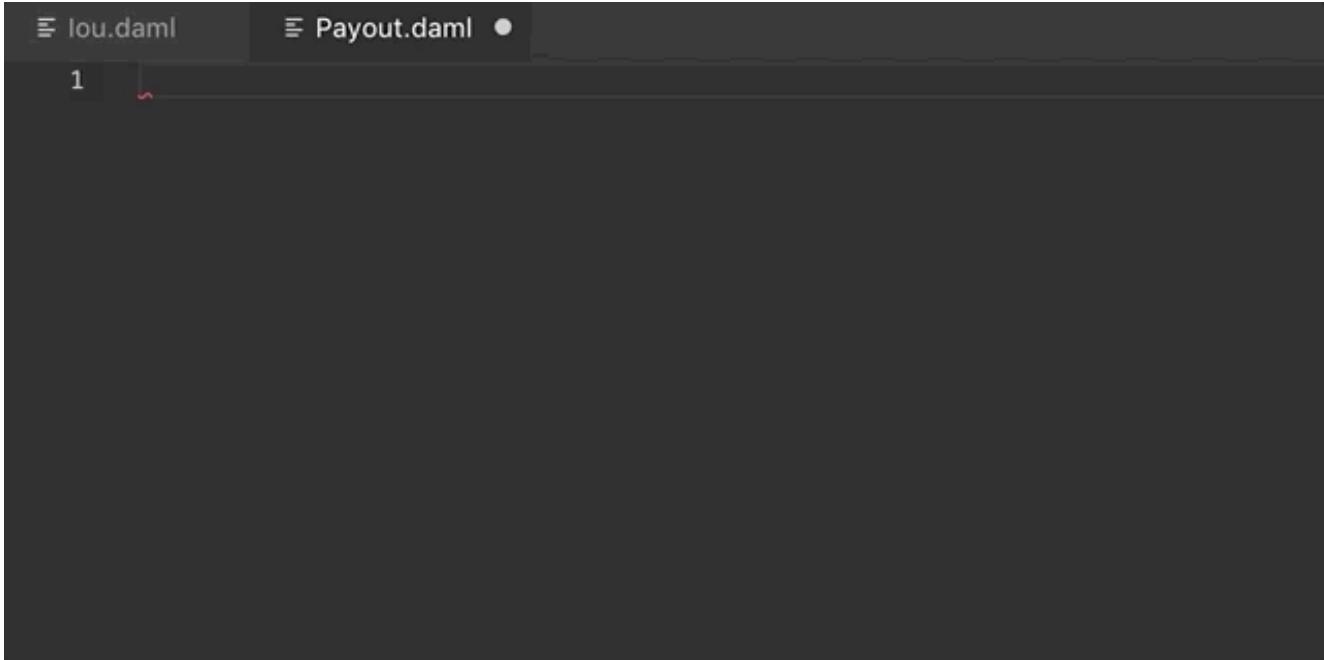
```
{
  "editor.tabCompletion": true,
```

(continues on next page)

(continued from previous page)

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

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



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

### 2.3.3 Common scenario errors

During DAML execution, errors can occur due to exceptions (e.g. use of abort, or division by zero), or due to authorization failures. You can expect to run into the following errors when writing DAML.

When a runtime error occurs in a scenario execution, the scenario result view shows the error together with the following additional information, if available:

**Last source location** A link to the last source code location encountered before the error occurred.

**Environment** The variables that are in scope when the error occurred. Note that contract identifiers are links that lead you to the transaction in which the contract was created.

**Ledger time** The ledger time at which the error occurred.

**Call stack** Call stack shows the function calls leading to the failing function. Updates and scenarios that do not take parameters are not included in the call stack.

**Partial transaction** The transaction that is being constructed, but not yet committed to the ledger.

**Committed transaction** Transactions that were successfully committed to the ledger prior to the error.

#### 2.3.3.1 Abort, assert, and debug

The `abort`, `assert` and `debug` inbuilt functions can be used in updates and scenarios. All three can be used to output messages, but `abort` and `assert` can additionally halt the execution:

```
abortTest = scenario do
    debug "hello, world!"
    abort "stop"
```

Scenario execution failed:  
 Aborted: stop  
 Ledger time: 1970-01-01T00:00:00Z  
 Partial transaction:  
 Trace:  
 "hello, world!"

### 2.3.3.2 Missing authorization on create

If a contract is being created without approval from all authorizing parties the commit will fail. For example:

```
template Example
  with
    party1 : Party; party2 : Party
  where
    signatory party1
    signatory party2

example = scenario do
  alice <- getParty "Alice"
  bob <- getParty "Bob"
  submit alice (create Example with party1=alice; party2=bob)
```

Execution of the example scenario fails due to ‘Bob’ being a signatory in the contract, but not authorizing the create:

Scenario execution failed:  
 #0: create of CreateAuthFailure:Example at unknown source  
 failed due to a missing authorization from 'Bob'  
 Ledger time: 1970-01-01T00:00:00Z  
 Partial transaction:  
 Sub-transactions:  
 #0  
 ↳ create CreateAuthFailure:Example  
 with  
 party1 = 'Alice'; party2 = 'Bob'

To create the Example contract one would need to bring both parties to authorize the creation via a choice, for example ‘Alice’ could create a contract giving ‘Bob’ the choice to create the ‘Example’ contract.

### 2.3.3.3 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.

#### 2.3.3.4 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.

## 2.4 Testing DAML using scenarios

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

Scenarios emulate the ledger. You can specify a linear sequence of actions that various parties take, and these are evaluated in order, according to the same consistency, authorization, and privacy rules as they would be on the sandbox ledger or ledger server. [DAML Studio](#) shows you the resulting Transaction graph.

For more on how scenarios work, see the [Examples](#) below.

On this page:

#### Scenario syntax

- [Scenarios](#)
- [Transaction submission](#)
- [Asserting transaction failure](#)
- [Full syntax](#)

#### Running scenarios in DAML Studio

##### Examples

- [Simple example](#)
- [Example with two updates](#)
- [Example with submitMustFail](#)

## 2.4.1 Scenario syntax

### 2.4.1.1 Scenarios

```
example =
  scenario do
```

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

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

### 2.4.1.2 Transaction submission

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

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

For example, a transaction could be [creating](#) a contract instance on the ledger, or [exercising](#) a choice on an existing contract.

### 2.4.1.3 Asserting transaction failure

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

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

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

#### 2.4.1.4 Full syntax

For detailed syntax, see [Reference: scenarios](#).

#### 2.4.2 Running scenarios in DAML Studio

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

#### 2.4.3 Examples

##### 2.4.3.1 Simple example

A very simple scenario looks like this:

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

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

##### 2.4.3.2 Example with two updates

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

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

      -- Alice exercises the contract, and receives payment.
```

(continues on next page)

(continued from previous page)

```
submit alice do
  exercise payAlice Call
```

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

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

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

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

Party "Bank of England" is assumed to know the definition of the `CallablePayout` contract template and the consequences of submitting a contract instance to the ledger.

Party "Alice" is assumed to know the definition of the contract template, as well as the consequences of exercising the `Call` choice on it. If "Alice" does not want to receive five pounds, she can simply not exercise that choice.

#### 2.4.3.3 Example with submitMustFail

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

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

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

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

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

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

## 2.5 Troubleshooting DAML

Error: <X> is not authorized to commit an update  
 Error Argument is not of serializable type  
 Modelling questions

- How to model an agreement with another party
- How to model rights
- How to void a contract
- How to represent off-ledger parties
- How to limit a choice by time
- How to model a mandatory action
- When to use Optional

Testing questions

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

### 2.5.1 Error: “<X> is not authorized to commit an update”

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

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

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

### 2.5.2 Error “Argument is not of serializable type”

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

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

Hovering over the compilation error displays:

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

## 2.5.3 Modelling questions

### 2.5.3.1 How to model an agreement with another party

To enter into an agreement, create a contract instance from a template that has explicit signatory and agreement statements.

You'll need to use a series of contracts that give each party the chance to consent, via a contract choice.

Because of the rules that DAML enforces, it is not possible for a single party to create an instance of a multi-party agreement. This is because such a creation would force the other parties into that agreement, without giving them a choice to enter it or not.

### 2.5.3.2 How to model rights

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

### 2.5.3.3 How to void a contract

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

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

### 2.5.3.4 How to represent off-ledger parties

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

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

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

### 2.5.3.5 How to limit a choice by time

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

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

### 2.5.3.6 How to model a mandatory action

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

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

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

### 2.5.3.7 When to use Optional

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

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

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

```
import DA.Optional
```

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

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

You can test for existence in various ways:

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

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

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

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

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

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

## 2.5.4 Testing questions

### 2.5.4.1 How to test that a contract is visible to a party

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

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

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

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

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

using an equality test and an assert:

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

### 2.5.4.2 How to test that an update action cannot be committed

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

## 2.6 Converting a project to DAML 1.2

In DA SDK version 0.11, we introduce DAML version 1.2. The SDK documentation and code examples all now refer to 1.2.

This page explains how to upgrade your DAML 1.0 and 1.1 code to DAML 1.2. It doesn’t explain what the new features in 1.2 are.

On this page:

**Introduction: the upgrade process****Start converting**

- 1. Upgrade to SDK version 0.11
- 2. Change language version to 1.2

**Templates**

- Separate lists of observers and signatories with commas
- Add appends (<>) between multi-line agreements
- Move template member functions outside the template

**Choices**

- Change how you specify a choice's return type
- Change how you introduce a choice
- Change anytime to nonconsuming
- Make choice names unique
- Stop re-using template argument names inside choices
- Change references to nested fields
  - \* Likely errors

**Scenarios**

- Change how you specify scenario tests
- Change commits and fails to submit and submitMustFail
- Replace Party literals with getParty
  - \* Likely errors

**Built-in types and functions**

- Replace Time literals with date and time
- Replace type names that have changed
- Remove Char
- Remove anonymous records
- Change Tuples
- Type of round function has changed
- Replace toText with show (except on Text)
- Replace fromInteger and toInteger
- Remove does

**Syntax**

- Change how you specify functions (def and fun)
- Change let

**Other changes**

- Change Standard Library imports
- Change Maybe
- Indentation rules are stricter
- Add deriving (Eq, Show) to any data type used in a template
  - \* Likely errors
- Fix warnings on importing modules where you don't use anything exported by it
- Update (non-DAML) application code
- Changes to privacy and authorization

**Other errors**

## 2.6.1 Introduction: the upgrade process

This page explains how to upgrade your DAML 1.0 and 1.1 code to DAML 1.2. It's organised into groups to make it easier to follow, but you don't necessarily need to follow the steps in order (apart from the steps under Start converting).

The list of changes here isn't comprehensive, but we've tried to cover as many things as possible, to help you upgrade.

A recommended approach is to start by converting files with no dependencies, resolving DAML Studio errors until they are all fixed. When you've fixed a file, you can then look at the files that depend on it, and so on upwards.

## 2.6.2 Start converting

### 2.6.2.1 1. Upgrade to SDK version 0.11

If you haven't already, upgrade your project to use SDK version 0.11. For details on how to do this, see [Managing SDK releases](#).

### 2.6.2.2 2. Change language version to 1.2

In all of your DAML files, change the version to 1.2. You need to do this for every DAML file in your codebase: you can't have a mix of versions within a project.

## 2.6.3 Templates

### 2.6.3.1 Separate lists of observers and signatories with commas

Previously, lists of `observer` and `signatory` parties could be separated by semicolons or commas. Now, you can only use commas.

Before:

```
signatory issuer; owner
```

After:

```
signatory issuer, owner
```

### 2.6.3.2 Add appends (<>) between multi-line agreements

In DAML 1.2, agreements are no longer a series of lines, but a one-line statement. If you have agreements that span multiple lines, you now need to concatenate them with `<>`.

Before:

```
agreement
  "REPO agreement was breached by the "
    toText seller.account.owner.agent
```

After:

```
agreement
  "REPO agreement was breached by the " <>
    show seller.account.owner.agent
```

### 2.6.3.3 Move template member functions outside the template

Previously, templates could have member functions declared using `def`. Now, you preface these using `let`.

Before:

```
template Foo
with
  p : Party
  msg : Text
where
  signatory p

  def crFoo (msg : Msg) =
    create Foo with msg; p
```

After:

```
template Foo
with
  p : Party
  msg : Text
where
  signatory p

  let crFoo _this msg =
    create Foo with msg; p = _this.p
```

## 2.6.4 Choices

### 2.6.4.1 Change how you specify a choice's return type

Previously, you used the keyword `returning` to specify a choice's return type. Now, use a `:` (semi-colon) after the choice name.

Before:

```
ChoiceName
with exampleArgument : ArgType
returning RetType
```

After:

```
ChoiceName : RetType
with exampleArgument : ArgType
```

### 2.6.4.2 Change how you introduce a choice

Previously, choice bodies were introduced with `to` (then, often, a `do`). Now, use `do`.

Before:

```
ChoiceName
  with exampleArgument : ArgType
  returning RetType
  to do action
```

After:

```
ChoiceName
  : RetType
  with exampleArgument : ArgType
  do action
```

#### 2.6.4.3 Change anytime to nonconsuming

The keyword `anytime` has been renamed to `nonconsuming`.

Before:

```
controller operator can
anytime InviteParticipant
```

After:

```
controller operator can
nonconsuming InviteParticipant
```

#### 2.6.4.4 Make choice names unique

Previously, different templates could duplicate choice names. Now, choice names in the same module must now be unique. For example, you can't have two `Accept` choices on different templates.

We recommend adding the template name to the start of the choice.

Before:

```
controller operator can Accept
```

After:

```
controller operator can Cash_Accept
```

#### 2.6.4.5 Stop re-using template argument names inside choices

Previously, you could shadow a template's parameter names, using the same names for parameters to choices. In DAML 1.2, this is no longer permitted.

Consequently you can no longer reuse an argument name from template in one of its' choices.

Before:

```
template Foo
with
  p : Party
  msg : Text
where
  signatory p
  controller p can ExampleChoice
    with msg : Text
```

After:

```
template Foo
with
  p : Party
  msg : Text
where
  signatory p
  controller p can ExampleChoice
    with otherMsg : Text
```

#### 2.6.4.6 Change references to nested fields

Previously, you could do nested field punning. Now, you need to explicitly assign the field.

Before:

```
create repoAgreement with
  r.buyer; r.seller; registerTime
```

After:

```
create repoAgreement with
  buyer = r.buyer
  seller = r.seller; registerTime
```

#### Likely errors

If you haven't made this change yet, you might see error messages like name not in scope, duplicate name, let in a do block, and indentation errors. For example:

```
/full/path/...daml:12:24: error:
  Not in scope: `x'
/full/path/...daml:12:31: error:
  Multiple declarations of `baz'
  Declared at: ....daml:12:32
                ....daml:12:31
```

## 2.6.5 Scenarios

### 2.6.5.1 Change how you specify scenario tests

Previously, scenarios were introduced using `test`. You can now remove `test`.

Before:

```
test myTest = scenario
  action
```

After:

```
myTest = scenario do
  action
```

### 2.6.5.2 Change commits and fails to submit and submitMustFail

`commits` has been renamed to `submit`, and `fails` has been renamed to `submitMustFail`. The ordering of the arguments has also changed.

Before:

```
partyA commits updateB

partyA fails updateB
```

After:

```
submit partyA do updateB
partyA `submit` updateB -- as an alternative

submitMustFail partyA do updateB
```

### 2.6.5.3 Replace Party literals with getParty

Party literals have been removed. Instead, use `getParty` to create a value of type `Party`.

Note that what can be in the party text is more limited now. Only alphanumeric characters, `-`, `_` and spaces.

Before:

```
let exampleParty = 'Example' party'
```

After:

```
exampleParty <- getParty "Example party"
```

Why was this change made? To ensure (using the type system) that a party could only be introduced in the context of a Scenario.

For party literals in choices, see [Remove does](#).

## Likely errors

If you haven't made this change yet, you might see error messages like:

```
/the/full/Path/To/MyFile.daml:12:13: error:
  * Syntax error on 'Example party'
    Perhaps you intended to use TemplateHaskell or TemplateHaskellQuotes
  * In the Template Haskell quotation 'Example party'
```

If the party has only one character, like 'D':

```
...:12:13: error:
  * Couldn't match expected type `Party'
            with actual type `GHC.Types.Char'
  * In the first argument of `submit', namely 'D'
    In the expression: 'D' `submit` assert $ 1 == 0
    In an equation for `wrongTest':
      wrongTest = 'D' `submit` assert $ 1 == 0
```

## 2.6.6 Built-in types and functions

### 2.6.6.1 Replace Time literals with date and time

Time literals have been removed. Instead, use the functions `date` and `time`.

Before:

```
exampleDate = 1970-01-02T00:00:00Z
```

After:

```
exampleDate = date 1970 Jan 1
exampleDateTime = time 0 0 0 exampleDate
```

### 2.6.6.2 Replace type names that have changed

`Integer` is now `Int`.  
`List` is now `[Type]`; `cons` is now `\::`; `nil` is now `[]`.  
`Empty value {}` is now `()`.

Before:

```
Integer

cons 1 (cons 2 nil) : List Integer

{ }
```

After:

```
Int
```

```
(1 :: 2 :: []) : [Int]  
()
```

### 2.6.6.3 Remove Char

In DAML 1.2, the primitive type `Char` and all of its related functions have been removed. Use `Text` instead.

Before:

```
-- Character literal for the character a  
'a'
```

After:

```
-- Now, use Text instead  
"a"
```

### 2.6.6.4 Remove anonymous records

You can no longer have anonymous *records* - for example, returned by choices, or as choice arguments.

Instead, use a pair/tuple, or define the data type outside the template.

Before:

```
controller owner can  
Split with splitAmount : Decimal  
    returning {  
        splitCid : ContractId Iou;  
        restCid : ContractId Iou  
    }
```

After:

```
controller owner can  
Split : (ContractId Iou, ContractId Iou)  
    with splitAmount : Decimal
```

Or, alternatively, define the data type outside the template:

```
data TwoCids = TwoCids with  
    splitCid : ContractId Iou  
    restCid : ContractId Iou
```

### 2.6.6.5 Change Tuples

A Tuple no longer has fields `fst` and `snd`. Instead use `_1`, `_2`, etc for all Tuples.

`Tuple3` is now `(,,)`.

Before:

```
def tsum (t : Tuple2 Integer Integer)
  : Integer =
  t.fst + t.snd
```

After:

```
tsum : (Int, Int) -> Int
tsum t = t._1 + t._2

-- or, you could do
tsum t = fst t + snd t
```

### 2.6.6.6 Type of round function has changed

The type of the `round` function has changed. Replace it with `roundBankers` or `roundCommercial`, depending on the rounding mode you want to use.

Before:

```
-- using bankers' rounding mode
round : Integer -> Decimal -> Decimal
```

After:

```
-- rounds away from zero
round : Decimal -> Int
```

### 2.6.6.7 Replace `toText` with `show` (except on Text)

Where you previously used `toText`, now use `show`.

However, note that the behaviour of `show` when applied to a `Text` value is different to the behaviour of `toText`, because `show` surrounds the value with Haskell string quotes.

Before:

```
> toText "bla"
"bla"
```

After:

```
> show "bla"
""bla""
```

### 2.6.6.8 Replace fromInteger and toInteger

Where you previously used `fromInteger`, now use `intToDecimal`.

Where you previously used `toInteger`, depending on what you want to achieve, use one of:

```
truncate (truncate x rounds x toward zero - closest to the behaviour of toInteger)
round (round to nearest integer, where a .5 is rounded away from zero)
floor (round down to nearest integer)
ceiling (round up to nearest integer)
```

### 2.6.6.9 Remove does

`does` has been removed, and you should remove it alongside the `Party` doing the action: this is now inferred.

Before:

```
party does action
```

After:

```
action
```

## 2.6.7 Syntax

### 2.6.7.1 Change how you specify functions (def and fun)

The `def` keyword has been removed. You don't need to replace it with anything.

You now separate signatures from function definitions.

The `fun` keyword for lambda expressions has been removed. Instead, use `\` (backslash).

Before:

```
def funcName (arg1 : Text) (arg2 : Integer) : Text = ...
def x = 1
map (fun x -> x * x) [1, 2, 3]
```

After:

```
funcName : Text -> Int -> Text
funcName arg1 arg2 = ...

x = 1

map (\x -> x * x) [1, 2, 3]
```

### 2.6.7.2 Change let

let in update and do blocks is unchanged; it still works like this:

```
do
  let x = 1
  pure x + 40
```

let in expressions has changed. It now requires the `in` keyword to specify wherein the bindings apply.

Before:

```
def foo (x : Integer) : Integer =
  let y = 40
  x + y
```

After:

```
foo : Int -> Int
foo x =
  let y = 40
  in x + y
```

### 2.6.8 Other changes

#### 2.6.8.1 Change Standard Library imports

Standard library modules are now organized slightly differently: `DA.<Something>` rather than `DA.Base.<Something>`.

Also, the standard library prelude now includes many functions that previously would need to be imported explicitly.

Before:

```
import DA.Base.List
import DA.Base.Map
```

After:

```
import DA.List
import DA.Map
```

#### 2.6.8.2 Change Maybe

Maybe has been renamed to Optional.

#### 2.6.8.3 Indentation rules are stricter

Previous versions of DAML were quite permissive with how much indentation you needed to add. These rules are now stricter.

### 2.6.8.4 Add deriving (Eq, Show) to any data type used in a template

Data types used in a template are now required to support classes Eq and Show.

Before:

```
data Bar = Bar
  with
    x : Integer
```

After:

```
data Bar = Bar
  with
    x : Integer
  deriving (Eq, Show)
```

#### Likely errors

If you haven't made this change yet, you might see error messages like:

```
* No instance for (Eq Bar)
  arising from the second field of `Foo` (type `Bar`)
  Possible fix:
    use a standalone 'deriving instance' declaration,
    so you can specify the instance context yourself
* When deriving the instance for (Eq Foo) DAML to Core
```

### 2.6.8.5 Fix warnings on importing modules where you don't use anything exported by it

Importing a module where you do not use anything exported by it now gives a warning. You can fix this warning by importing using `import Module()`.

Before:

```
import Foo
```

After:

```
import Foo()
```

### 2.6.8.6 Update (non-DAML) application code

There is a change as a consequence of DAML 1.2 in how applications exercise choices with no parameters: they're now exercised in a way that's more similar to choices with parameters.

If you have an application (based on GRPC, Java binding, or something else) that exercises any choices with no parameters, you need to update that code.

In the DAML-LF produced from DAML 1.1, choices with no parameters take a Unit argument. Now, they take an argument of type data Foo = Foo, where Foo is the choice name. This is a record with zero fields.

#### 2.6.8.7 Changes to privacy and authorization

The behavior around what parties have access to has changed in two ways:

There used to be a situation where parties could get access to contracts that they were not stakeholders to.

This happened for templates that have a contract ID as an argument to the template.

Previously, when a contract was created from such a template, the contract ID passed as an argument (and so the contents of the contract) were disclosed to all parties that could see the new contract. This is no longer the case in DAML 1.2.

The current behavior applies to all templates. Now, when a choice is exercised on a contract, that contract's ID (and so the contract's contents) are disclosed to all parties that can see the exercise.

The reasoning behind this is to make sure that all parties seeing the exercise can recompute and confirm the result of exercising that choice.

Previously, if a party could get hold of a contract ID, they could fetch the contract that ID related to.

Now, fetches only succeed when at least one party that is a stakeholder of the fetched contract authorizes the fetch. Otherwise the fetch will fail.

This means that a party can fetch a contract if and only if a stakeholder of that contract authorized the disclosure to that party.

#### 2.6.9 Other errors

This conversion guide is not comprehensive. If you can't work out how to convert something, please [get in touch](#).

### 2.7 Writing good DAML

#### 2.7.1 DAML design patterns

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

[Download all the example code](#)

**Initiate and Accept** The Initiate and Accept pattern demonstrates how to start a bilateral workflow.

One party initiates by creating a proposal or an invite contract. This gives another party the chance to accept, reject or renegotiate.

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

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

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

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

### 2.7.1.1 Initiate and Accept

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

#### Motivation

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

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

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

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

#### Implementation

The Initiate and Accept pattern in general involves 2 contracts:

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

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

    controller issuer can
      nonconsuming Invite : ContractId CoinIssueProposal
```

(continues on next page)

(continued from previous page)

```
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 = []
```

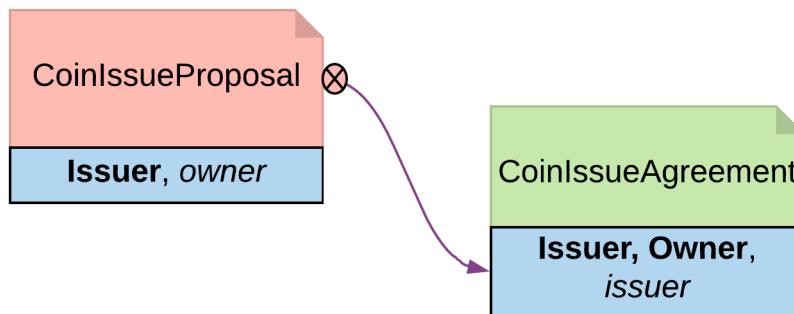


Fig. 2: Initiate and Accept pattern diagram

## Trade-offs

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

### 2.7.1.2 Multiple party agreement

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

## Motivation

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

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

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

## Implementation

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

```
template Agreement
  with
    person1: Party
    person2: Party
    person3: Party
    person4: Party
    agree: Text
  where
    signatory person1, person2, person3, person4
    agreement agree
```

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

The Pending contract has the same list of signatories as the Agreement contract. Each party of the Agreement has a choice(s) to add their signature.

```
template Pending
with
    agree: Agreement
    person1: Party
    person2: Party
    person3: Party
    person4: Party
where
    signatory person1, person2, person3, person4

    controller agree.person2 can
        Sign2 : ContractId Pending
        do
            create this with person2 = agree.person2
    controller agree.person3 can
        Sign3 : ContractId Pending
        do create this with person3 = agree.person3
    controller agree.person4 can
        Sign4 : ContractId Pending
        do create this with person4 = agree.person4
```

One of the stakeholders acts as the coordinator, and has a choice to create the final Agreement contract once all parties have signed.

```
controller person1 can
    Finalize : ContractId Agreement
    do
        assert (person1 == agree.person1 && person2 == agree.person2
                && person3 == agree.person3 && person4 == agree.person4)
        create agree with
            person1; person2; person3; person4
```

**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 itself in all signatory slots.

```
[person1, person2, person3, person4] <- makePartiesFrom ["Alice",
->"Bob", "Clare", "Dave"]
let agree = Agreement with person1; person2; person3; person4; agree
->= "Have a party"

agree <- person1 `submit` do
    create Pending with agree; person1; person2 = person1; person3 =
->person1; person4 = person1
```

Once the Pending contract is created, the other parties can exercise choices to Accept, Reject, or Negotiate. For simplicity, the example code only has choices to express consensus.

```
agree <- person2 `submit` do exercise agree Sign2
agree <- person3 `submit` do exercise agree Sign3
```

(continues on next page)

(continued from previous page)

```
agree <- person4 `submit` do exercise agree Sign4
```

The coordinating party can create the Agreement contract on the ledger, and finalize the process when all parties have signed and agreed to the multi-party agreement.

```
person1 `submit` do exercise agree Finalize
```

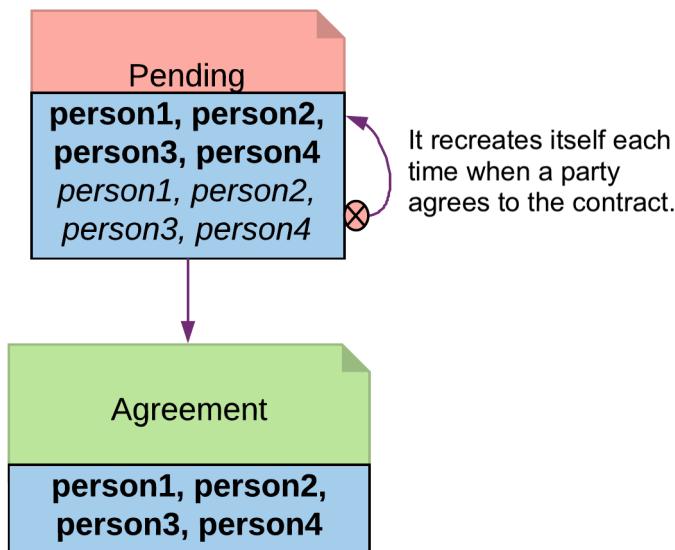


Fig. 3: Multiple Party Agreement Diagram

**Note:** DA is developing new DAML features for dynamic checks on unbounded explicit signatures. This would obviate the Multiple Party Agreement pattern altogether.

### 2.7.1.3 Delegation

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

#### Motivation

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

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

## Implementation

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

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

  controller owner can

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

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

    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
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

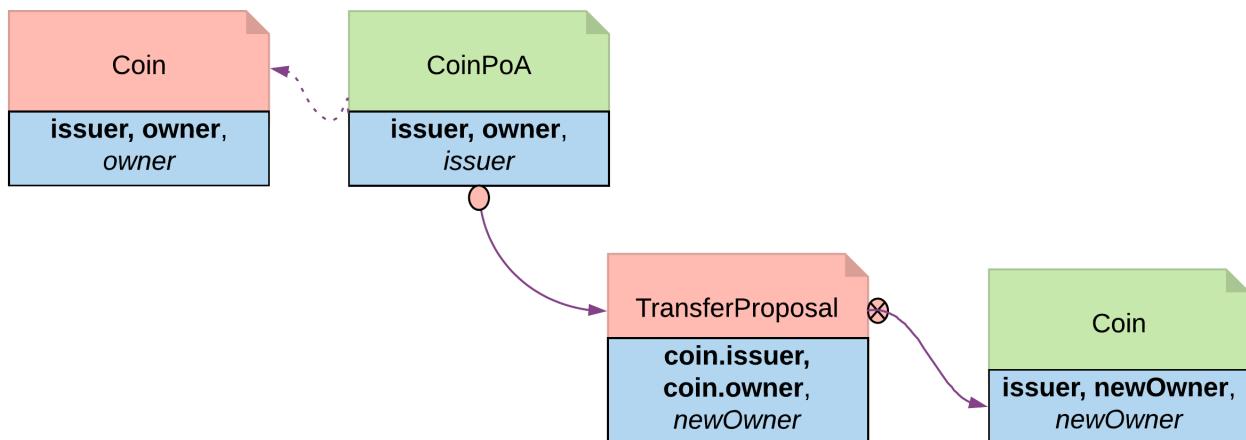


Fig. 4: Delegation pattern diagram

### 2.7.1.4 Authorization

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

#### Motivation

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

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

#### Authorization

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

### 2.7.1.5 Locking

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

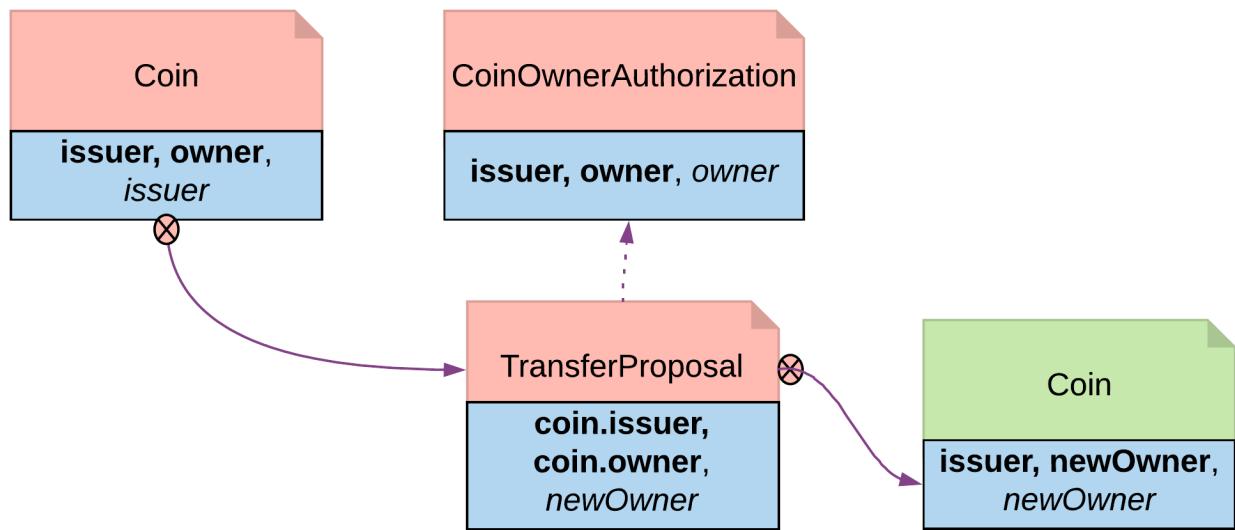


Fig. 5: Authorization Diagram

## Motivation

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

## Implementation

There are three ways to achieve locking:

### Locking by archiving

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

```

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

(continues on next page)

(continued from previous page)

```

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

```

(continues on next page)

(continued from previous page)

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

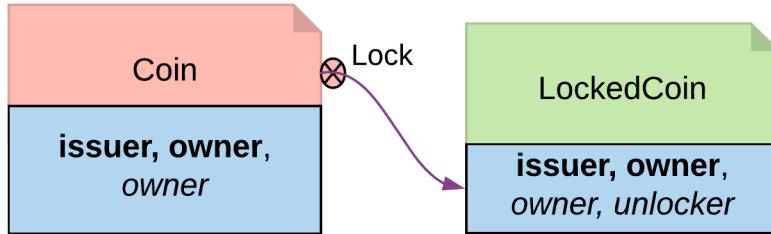


Fig. 6: Locking By Consuming Choice Diagram

### Archiving contract

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

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

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

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

```
template CoinCommitment
  with
    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
```

(continues on next page)

(continued from previous page)

```

: 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
→archived
  prop <- exercise coinCid Transfer with newOwner = issuer
  do
    id <- exercise prop AcceptTransfer
    exercise id Archives
    --create a lockedCoin to represent the coin in locked state
    create LockedCoin with
      coin=inputCoin with owner; issuer; amount
      maturity; locker

```

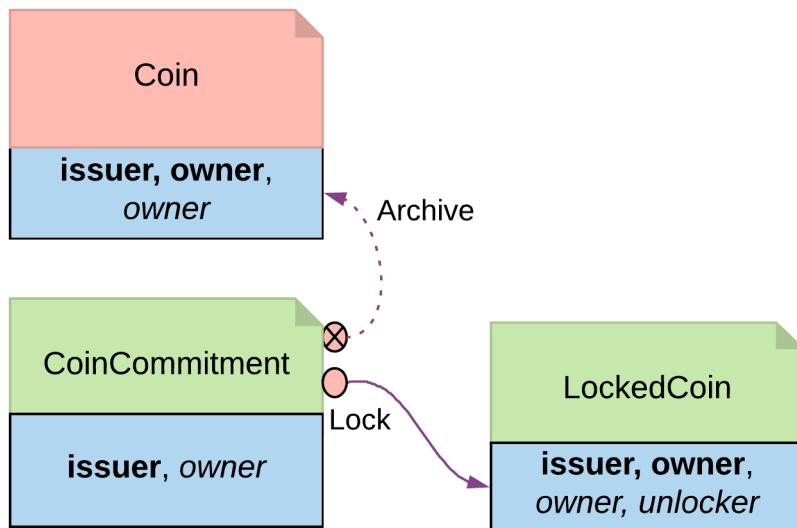


Fig. 7: Locking By Archiving Contract Diagram

## Trade-offs

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

Locking by archiving disables all choices on the original contract. Usually for consuming choices this is exactly what is required. But if a party needs to selectively lock only some choices, remaining active choices need to be replicated on the *LockedCoin* contract, which can lead to code duplication.

The choices on the original contract need to be altered for the lock choice to be added. If this contract is shared across multiple participants, it will require agreement from all involved.

## Locking by state

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

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

  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
```

(continues on next page)

(continued from previous page)

```

owner: Party
issuer: Party
amount: Decimal
locker: Party

where
  signatory issuer
  signatory owner

ensure amount > 0.0

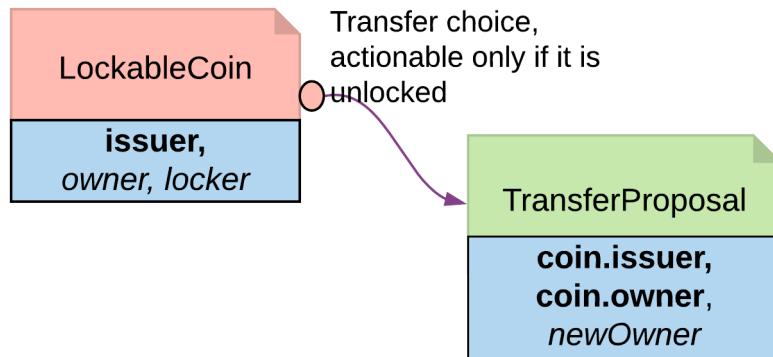
--Transfer can happen only if it is not locked
controller owner can
  Transfer : ContractId TransferProposal
    with newOwner: Party
    do
      assert (locker == owner)
      create TransferProposal
        with coin=this; newOwner

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

--Unlock only makes sense if the coin is in locked state
controller locker can
  Unlock
    : ContractId LockableCoin
    do
      assert (locker /= owner)
      create this with locker = owner

```

## Locking By State Diagram



## Trade-offs

It requires changes made to the original contract template. Furthermore you should need to change all choices intended to be locked.

If locking and unlocking terms (e.g. lock triggering event, expiry time, etc) need to be added to the template parameters to track the state change, the template can get overloaded.

## Locking by safekeeping

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

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

controller owner can

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

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

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

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

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

```

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

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

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

```

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

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

(continues on next page)

(continued from previous page)

```
tpCid <- exercise coinCid Transfer with newOwner = coin.owner
exercise tpCid AcceptTransfer
```

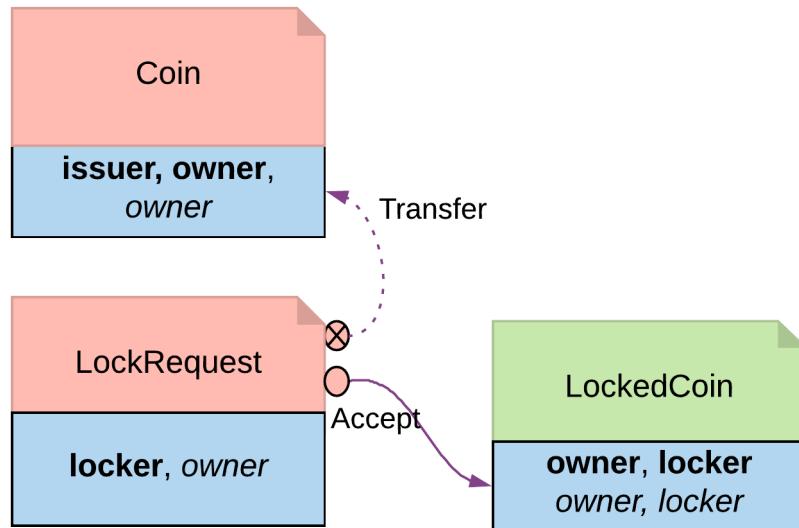


Fig. 8: Locking By Safekeeping Diagram

## Trade-offs

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

### 2.7.1.6 Diagram legends

### 2.7.2 DAML anti-patterns

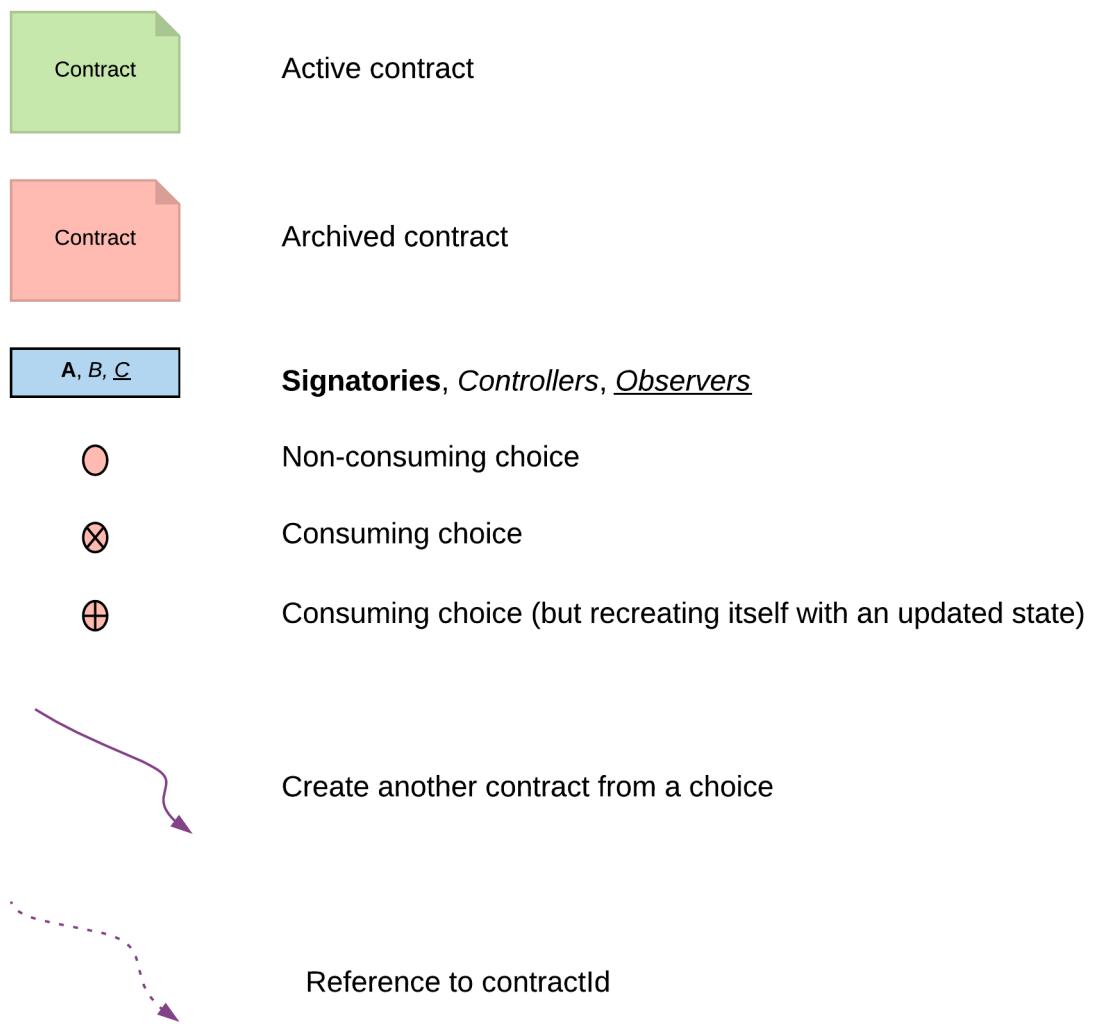
This documents DLT anti-patterns, their drawbacks and more robust ways of achieving the same outcome.

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

#### 2.7.2.1 Don't use the ledger for orchestration

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

Committing assets to a settlement cycle at 10:00 am



Starting a netting calculation after trade registration has finished  
 Triggering the optimization of a portfolio

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

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

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

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

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

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

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

### 2.7.2.2 Avoid race conditions in smart contracts

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

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

(continues on next page)

(continued from previous page)

```

controller payer can
  PayerAllocate: ContractId DvP
-- 

controller receiver can
  ReceiverAllocate: ContractId DvP
-- 

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

```

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

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

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

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

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

### 2.7.2.3 Don't use status variables in smart contracts

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

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

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

```

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

template Obligation

```

(continues on next page)

(continued from previous page)

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

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

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

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

The drawbacks of maintaining status variables in contracts are:

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

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

### 2.7.3 What functionality belongs in DAML models versus application code?

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

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

<a href="#">Looking at the ledger from a legal perspective</a> <a href="#">Looking at the ledger from a data synchronization perspective</a>
---

### 2.7.3.1 Looking at the ledger from a legal perspective

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

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

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

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

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

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

### 2.7.3.2 Looking at the ledger from a data synchronization perspective

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

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

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

## Chapter 3

# Building Applications

### 3.1 Building applications against a DA ledger

This document is a guide to building applications that interact with a DA ledger deployment (the ‘ledger’).

It describes the characteristics of the ledger API, how this affects the way an application is built (the application architecture), and why it is important to understand this when building applications. It then describes the resources provided by Digital Asset to help with this task, and some guidelines that can help you build correct, performant, and maintainable applications using all of the supported languages.

#### 3.1.1 Categories of application

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

Category	Receives transactions?	Sends commands?	Example
source	No	Yes	An injector that reads new contracts from a file and injects them into the system.
sink	Yes	No	A reader that pipes data from the ledger into an SQL database.
automation	Yes	Yes, responding to transactions	Automatic trade registration.
interactive	Yes (and displays to user)	Yes, based on user input	DA’s <a href="#">Navigator</a> , which lets you see and interact with the ledger.

Additionally, applications can be written in two different styles:

Event-driven - applications base their actions on individual ledger events only.

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

### 3.1.1.1 Event-driven applications

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

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

### 3.1.1.2 State-driven applications

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

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

### 3.1.1.3 Which approach to take

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

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

## 3.1.2 The Ledger API

All applications interact with the ledger through a well defined API - the **Ledger API**. This has some characteristics that are important to understand when designing an application.

The ledger platform itself has been designed around some specific architectural ideas, primarily to enable a high degree of horizontal scalability. This architecture, called Command-Query Responsibility Separation ([CQRS](#)), causes the API to be structured as two separate data streams:

A stream of **commands** TO the platform that allow an application to cause state changes.

A stream of **events** FROM the platform that indicate all state changes taking place on the platform.

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.

The API itself is implemented as a set of services and messages, defined as [gRPC](#) protocol definitions and implemented using the [gRPC](#) and [protobuf](#) compiler. These are layered on top of an HTTP/2 transport implemented by the platform servers. Full details can be found in the [Ledger API Introduction](#)

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 submission that cause them.

The application must handle these issues, and is a major determinant of application architecture. Understanding the consequences of the API characteristics is important for a successful application design.

This document is intended to help you understand these issues so you can build correct, performant and maintainable applications.

### 3.1.2.1 API services

The API is structured as a set of services, implemented using [gRPC](#) and [Protobuf](#). For detailed information, see the [reference documentation for services and structure](#).

Most applications will not use this low-level API, instead using the language bindings (e.g. [Java](#), [Javascript](#)). These bindings provide programming-language specific access to the API, and often implement higher level APIs that provide additional, useful features to an application.

#### Ledger Services Layer

##### Command Submission Service

Use the Command Submission service to submit a command to the ledger. Commands either create a new template 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 syntactical or content reasons.

The on-ledger effect of the command execution will be reported via an event delivered by 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.

Commands can be labeled with two application-specific ID's, a [commandId](#), and a [workflowId](#), and both are returned in completion events. The [commandId](#) is returned to the submitting application only, and is generally used to implement this correlation between commands and completions. The [workflowId](#) is also returned (via a transaction event) to all applications receiving transactions resulting from a command. This can be used to track commands submitted by other applications.

##### 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 its model of commands in flight, and implement any application-specific error recovery. See [Common Tasks](#) below for more details.

## Transaction Service

Use the Transaction Service to listen to changes in the ledger state, reported via a stream of transaction events.

Transaction events detail the changes on transaction boundaries - each event denotes a transaction on the ledger, and contains all the update events (create, exercise, archive of contracts) that had an effect in that transaction.

Transaction events contain a [transactionId](#) (assigned by the server), the [workflowId](#), the [commandId](#), and the events in the transaction.

Transaction events are the primary mechanism by which an application will do its work. Event-driven applications can use them to generate new commands, and state-driven applications will use them to update their state model, by e.g. creating data that represents created contracts.

The Transaction Service can be initiated to read events from an arbitrary point on the ledger. This is important when starting or restarting an application, and works in conjunction with the [Active Contract service](#)

## Package Service

Use the Package Service to obtain information about DAML programs and packages loaded into the server.

This is useful for obtaining type and metadata information that allow you to interpret event data in a more useful way.

## Ledger Identity Service

Use the Ledger Identity service to obtain the identity string of the ledger that it is connected to.

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

## Ledger Configuration Service

Use the Ledger Configuration Service to subscribe to changes in ledger configuration.

This configuration includes maximum and minimum values for the difference in Ledger Effective Time and Maximum Record Time (see [Time Service](#) for details of these).

## Time Service

Use the Time Service to obtain the time as known by the ledger server.

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

MRT is used in the detection of lost commands. See [Common Tasks](#) for more detail.

## Application Services Layer

### 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 of 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 allows simple applications to be completely stateless, and alleviates the need for them to track command submissions.

### Active Contract Service

Use the Active Contract Service to obtain a party-specific view of all the contracts recently active on the ledger.

The Active Contract Service returns the current contract set as a set of created events that would re-create the state being reported, along with the ledger position at which the view of the set was taken.

For state-driven applications, this is most important at application start. They must synchronize their initial state with a known view of the ledger, and without this service, the only way to do this would be to read the Transaction Stream from the beginning of the ledger. This can be prohibitive with a large ledger.

The Active Contract Service overcomes this, by allowing an application to request a snapshot of the ledger, determine the position at which that snapshot was taken, and build its initial state from this view. The application can then begin to receive events via the Transaction Service from the given position, and remain in sync with the ledger by using these to apply updates to this initial state.

### 3.1.3 Structuring an application

Although applications that communicate with the ledger have many purposes, they generally have some common features, usually related to their style: event-driven or state-driven. This section describes these commonalities, and the major functions of each of these styles.

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

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

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

### 3.1.3.1 Event-driven applications

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

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

To do this, the application should:

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

### 3.1.3.2 State-driven applications

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

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

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

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

To do this, the application should:

1. Obtain the initial state of the ledger by using the Active Contract service, processing each event received to create an initial application state.
2. Create a connection to the Transaction Service to receive new events from that initial state, and instantiate a stream handler to process them.
3. Create a connection to the Command Submission Service to send commands.
4. Create a connection to the Command Completion Service, and set up a stream handler to handle completions.
5. Read the event stream and process each event to update its view of the ledger state.

To make accessing and examining this state easier, this often involves turning the generic description of create contracts into instances of structures (such as class instances that are

- more appropriate for the language being used. This also allows the application to ignore contract data it does not need.
6. Examine the state at regular intervals (often after receiving and processing each transaction event) and send commands back to the ledger on significant changes.
  7. Maintain a record of **pending contracts**: contracts that will be archived by these commands, but whose completion has not been received.  
Because of the asynchronous nature of the API, these contracts will not exist on the ledger at some point after the command has been submitted, but will exist in the application state until the corresponding archive event has been received. Until that happens, the application must ensure that these **pending contracts** are not considered part of the application state, even though their archives have not yet been received. Processing and maintaining this pending set is a crucial part of a state-driven application.
  8. Examine command completions, and handle any command errors. As well as application defined needs (such as command re-submission and de-duplications), this must also include handling command errors as described [Common Tasks](#), and also consider the pending set. Exercise commands that fail mean that contracts that are marked as pending will now not be archived (the application will not receive any archive events for them) and must be returned to the application state.

### 3.1.3.3 Common tasks

Both styles of applications will take the following steps:

Define an **applicationId** - this identifies the application to the ledger server.

Connect to the ledger (including handling authentication). This creates a client interface object that allows creation of the stream connection described in [Structuring an application](#).

Handle execution errors. Because these are received asynchronously, the application will need to keep a record of commands in flight - those sent but not yet indicated complete (via an event). Correlate commands and completions via an application-defined **commandId**. Categorize different sets of commands with a [workflowId](#).

Handle lost commands. The ledger server does not guarantee that all commands submitted to it will be executed. This means that a command submission will not result in a corresponding completion, and some other mechanism must be employed to detect this. This is done using the values of Ledger Effective Time (LET) and Maximum Record Time (MRT). The server does guarantee that if a command is executed, it will be executed within a time window between the LET and MRT specified in the command submission. Since the value of the ledger time at which a command is executed is returned with every completion, reception of a completion with a record time that is greater than the MRT of any pending command guarantees that the pending command will not be executed, and can be considered lost.

Have a policy regarding command resubmission. In what situations should failing commands be re-submitted? Duplicate commands must be avoided in some situations - what state must be kept to implement this?

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

### 3.1.4 Application Libraries

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

binding libraries.

### 3.1.4.1 Java

The Java API bindings have three levels:

A low-level Data Layer, including Java classes generated from the gRPC protocol definition files and thin layer of support classes. These provide a builder pattern for constructing protocol items, and blocking and non-blocking interfaces for sending and receiving requests and responses.

A Reactive Streams interface, exposing all API endpoints as [RxJava Flowables](#).

A Reactive Components API that uses the above to provide high-level facilities for building state-driven applications.

For more information on these, see the documentation: a [tutorial/description](#) and the [JavaDoc reference](#).

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

### 3.1.4.2 Scala

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

### 3.1.4.3 gRPC

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

## 3.1.5 Architecture Guidance

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

### 3.1.5.1 Use a reactive architecture and libraries

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

It matches well to the streaming nature of the ledger API.

It will handle all the multi-threading issues, providing you with sequential model to implement your application code.

It allows for several implementation strategies that are inherently scalable e.g RxJava, Akka Streams/Actors, RxJS, RxPy etc.

### 3.1.5.2 Prefer a state-driven approach

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

It's easier to add direct event handling to state-driven applications than the reverse.

Most applications have to keep some state.

DigitalAsset language bindings directly support the pattern, and provide libraries that handle many of the required tasks.

### 3.1.5.3 Consider a state-driven application as a function of state to commands

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

It separates the application into separate stages of event transformation, state update and command generation.

The command generation is the core of the application - implementing as a pure function makes it easy to reason about, and thus reduces bugs and fosters correctness.

Doing this will also require that the application is structured so that the state examined by that function is stable - that is, not subject to an update while the function is running. This is one of the things that makes the function, and hence the application, easier to reason about.

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

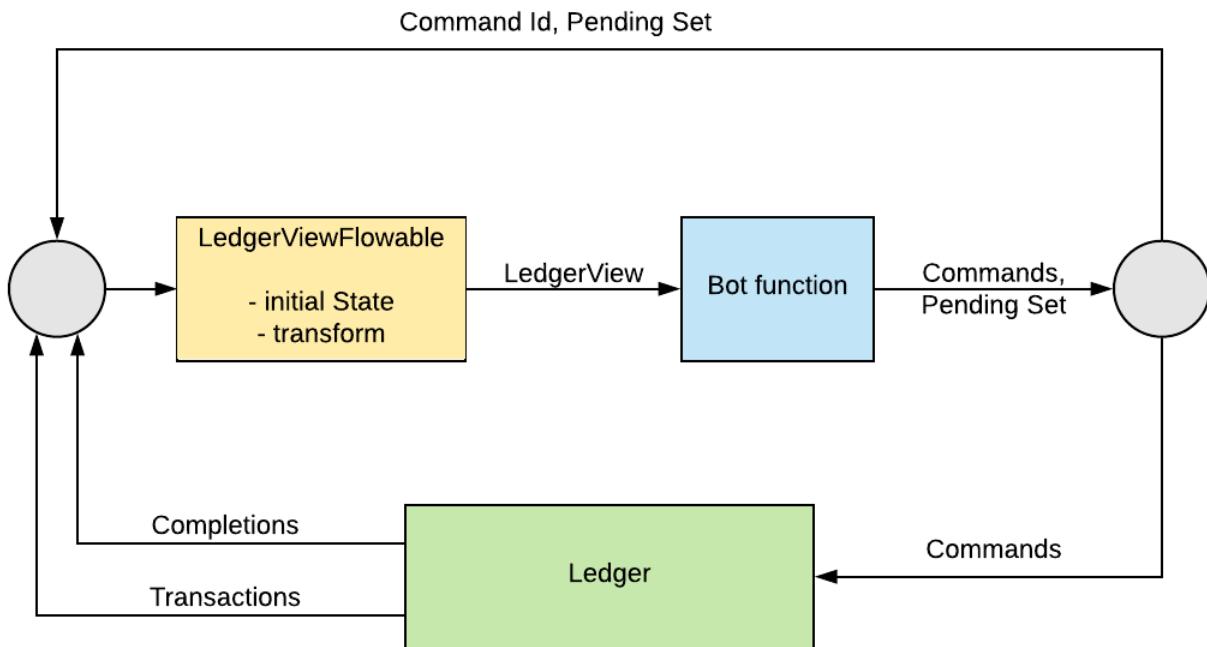
This allows you to reduce the work of building and application to the tasks of:

defining the Bot function.

defining the event transformation.

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

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



Full details of the framework are available in the links described in the [Java library](#) above.

## 3.2 Ledger API

A DAML Ledger provides an API to receive data from and send data to the ledger, subsequently called **Ledger API**. The API is separated into a small number of services that cover various aspects of the ledger, e.g. reading transactions or submitting commands. The Ledger API is defined using [Google Protocol Buffers](#) and [gRPC](#).

### 3.2.1 Ledger API Reference

#### Table of Contents

- [grpc/health/v1/health\\_service.proto](#)
  - [HealthCheckRequest](#)
  - [HealthCheckResponse](#)
  - [HealthCheckResponse.ServingStatus](#)
  - [Health](#)
- [google/rpc/code.proto](#)
  - [Code](#)
- [google/rpc/status.proto](#)
  - [Status](#)
    - \* [Overview](#)
    - \* [Language mapping](#)
    - \* [Other uses](#)
- [google/rpc/error\\_details.proto](#)
  - [BadRequest](#)

- [BadRequest.FieldViolation](#)
- [DebugInfo](#)
- [Help](#)
- [Help.Link](#)
- [LocalizedMessage](#)
- [PreconditionFailure](#)
- [PreconditionFailure.Violation](#)
- [QuotaFailure](#)
- [QuotaFailure.Violation](#)
- [RequestInfo](#)
- [ResourceInfo](#)
- [RetryInfo](#)

[com/digitalasset/ledger/api/v1/trace\\_context.proto](#)

- [TraceContext](#)

[com/digitalasset/ledger/api/v1/ledger\\_identity\\_service.proto](#)

- [GetLedgerIdentityRequest](#)
- [GetLedgerIdentityResponse](#)
- [LedgerIdentityService](#)

[com/digitalasset/ledger/api/v1/ledger\\_offset.proto](#)

- [LedgerOffset](#)
- [LedgerOffset.LedgerBoundary](#)

[com/digitalasset/ledger/api/v1/package\\_service.proto](#)

- [GetPackageRequest](#)
- [GetPackageResponse](#)
- [GetPackageStatusRequest](#)
- [GetPackageStatusResponse](#)
- [ListPackagesRequest](#)
- [ListPackagesResponse](#)
- [HashFunction](#)
- [PackageStatus](#)
- [PackageService](#)

[com/digitalasset/ledger/api/v1/transaction.proto](#)

- [Transaction](#)
- [TransactionTree](#)
- [TransactionTree.EventsByIdEntry](#)
- [TreeEvent](#)

[com/digitalasset/ledger/api/v1/completion.proto](#)

- [Completion](#)

[com/digitalasset/ledger/api/v1/transaction\\_service.proto](#)

- [GetLedgerEndRequest](#)
- [GetLedgerEndResponse](#)
- [GetTransactionByEventIdRequest](#)
- [GetTransactionByIdRequest](#)
- [GetTransactionResponse](#)
- [GetTransactionTreesResponse](#)
- [GetTransactionsRequest](#)
- [GetTransactionsResponse](#)
- [TransactionService](#)

[com/digitalasset/ledger/api/v1/value.proto](#)

- [Identifier](#)

- [List](#)
- [Optional](#)
- [Record](#)
- [RecordField](#)
- [Value](#)
- [Variant](#)

[com/digitalasset/ledger/api/v1/ledger\\_configuration\\_service.proto](#)

- [GetLedgerConfigurationRequest](#)
- [GetLedgerConfigurationResponse](#)
- [LedgerConfiguration](#)
- [LedgerConfigurationService](#)

[com/digitalasset/ledger/api/v1/testing/reset\\_service.proto](#)

- [ResetRequest](#)
- [ResetService](#)

[com/digitalasset/ledger/api/v1/testing/time\\_service.proto](#)

- [GetTimeRequest](#)
- [GetTimeResponse](#)
- [SetTimeRequest](#)
- [TimeService](#)

[com/digitalasset/ledger/api/v1/event.proto](#)

- [ArchivedEvent](#)
- [CreatedEvent](#)
- [Event](#)
- [ExercisedEvent](#)

[com/digitalasset/ledger/api/v1/command\\_completion\\_service.proto](#)

- [Checkpoint](#)
- [CompletionEndRequest](#)
- [CompletionEndResponse](#)
- [CompletionStreamRequest](#)
- [CompletionStreamResponse](#)
- [CommandCompletionService](#)

[com/digitalasset/ledger/api/v1/command\\_service.proto](#)

- [SubmitAndWaitRequest](#)
- [CommandService](#)

[com/digitalasset/ledger/api/v1/transaction\\_filter.proto](#)

- [Filters](#)
- [InclusiveFilters](#)
- [TransactionFilter](#)
- [TransactionFilter.FiltersByPartyEntry](#)

[com/digitalasset/ledger/api/v1/commands.proto](#)

- [Command](#)
- [Commands](#)
- [CreateCommand](#)
- [ExerciseCommand](#)

[com/digitalasset/ledger/api/v1/command\\_submission\\_service.proto](#)

- [SubmitRequest](#)
- [CommandSubmissionService](#)

[com/digitalasset/ledger/api/v1/active\\_contracts\\_service.proto](#)

- [GetActiveContractsRequest](#)
- [GetActiveContractsResponse](#)

- [ActiveContractsService](#)
- Scalar Value Types

### 3.2.1.1 grpc/health/v1/health\_service.proto

#### HealthCheckRequest

Field	Type	Label	Description
service	<a href="#">string</a>		

#### HealthCheckResponse

Field	Type	Label	Description
status	<a href="#">HealthCheckResponse.ServingStatus</a>		

#### HealthCheckResponse.ServingStatus

Name	Number	Description
UNKNOWN	0	
SERVING	1	
NOT_SERVING	2	

#### Health

Method Name	Request Type	Response Type	Description
Check	<a href="#">HealthCheckRequest</a>	<a href="#">HealthCheckResponse</a>	

### 3.2.1.2 google/rpc/code.proto

#### Code

The canonical error codes for Google APIs.

Sometimes multiple error codes may apply. Services should return the most specific error code that applies. For example, prefer `OUT_OF_RANGE` over `FAILED_PRECONDITION` if both codes apply. Similarly prefer `NOT_FOUND` or `ALREADY_EXISTS` over `FAILED_PRECONDITION`.

Name	Number	Description
OK	0	Not an error; returned on success HTTP Mapping: 200 OK
CANCELLED	1	The operation was cancelled, typically by the caller. HTTP Mapping: 499 Client Closed Request
UNKNOWN	2	Unknown error. For example, this error may be returned when a Status value received from another address space belongs to an error space that is not known in this address space. Also errors raised by APIs that do not return enough error information may be converted to this error. HTTP Mapping: 500 Internal Server Error
IN-VALID_ARGUMENT	3	The client specified an invalid argument. Note that this differs from FAILED_PRECONDITION. INVALID_ARGUMENT indicates arguments that are problematic regardless of the state of the system (e.g., a malformed file name). HTTP Mapping: 400 Bad Request
DEADLINE_EXCEEDED	4	The deadline expired before the operation could complete. For operations that change the state of the system, this error may be returned even if the operation has completed successfully. For example, a successful response from a server could have been delayed long enough for the deadline to expire. HTTP Mapping: 504 Gateway Timeout
NOT_FOUND	5	Some requested entity (e.g., file or directory) was not found. Note to server developers: if a request is denied for an entire class of users, such as gradual feature rollout or undocumented whitelist, NOT_FOUND may be used. If a request is denied for some users within a class of users, such as user-based access control, PERMISSION_DENIED must be used. HTTP Mapping: 404 Not Found
ALREADY_EXISTS	6	The entity that a client attempted to create (e.g., file or directory) already exists. HTTP Mapping: 409 Conflict
PERMISSION_DENIED	7	The caller does not have permission to execute the specified operation. PERMISSION_DENIED must not be used for rejections caused by exhausting some resource (use RESOURCE_EXHAUSTED instead for those errors). PERMISSION_DENIED must not be used if the caller can not be identified (use UNAUTHENTICATED instead for those errors). This error code does not imply the request is valid or the requested entity exists or satisfies other pre-conditions. HTTP Mapping: 403 Forbidden
UNAUTHENTICATED	16	The request does not have valid authentication credentials for the operation. HTTP Mapping: 401 Unauthorized
RESOURCE_EXHAUSTED	8	Some resource has been exhausted, perhaps a per-user quota, or perhaps the entire file system is out of space. HTTP Mapping: 429 Too Many Requests
FAILED_PRECONDITION	9	The operation was rejected because the system is not in a state required for the operation's execution. For example, the directory to be deleted is non-empty, an rmdir operation is applied to a non-directory, etc. Service implementors can use the following guidelines to decide between FAILED_PRECONDITION, ABORTED, and UNAVAILABLE: (a) Use UNAVAILABLE if the client can retry at a higher level (e.g., when a client-specified test-and-set fails, indicating the client should restart a read-modify-write sequence). (c) Use FAILED_PRECONDITION if

### 3.2.1.3 google/rpc/status.proto

#### Status

The `Status` type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by [gRPC](#). The error model is designed to be:

- Simple to use and understand for most users
- Flexible enough to meet unexpected needs

#### Overview

The `Status` message contains three pieces of data: error code, error message, and error details. The error code should be an enum value of [google.rpc.Code](#), but it may accept additional error codes if needed. The error message should be a developer-facing English message that helps developers understand and resolve the error. If a localized user-facing error message is needed, put the localized message in the error details or localize it in the client. The optional error details may contain arbitrary information about the error. There is a predefined set of error detail types in the package `google.rpc` that can be used for common error conditions.

#### Language mapping

The `Status` message is the logical representation of the error model, but it is not necessarily the actual wire format. When the `Status` message is exposed in different client libraries and different wire protocols, it can be mapped differently. For example, it will likely be mapped to some exceptions in Java, but more likely mapped to some error codes in C.

#### Other uses

The error model and the `Status` message can be used in a variety of environments, either with or without APIs, to provide a consistent developer experience across different environments.

Example uses of this error model include:

- Partial errors. If a service needs to return partial errors to the client, it may embed the `Status` in the normal response to indicate the partial errors.
- Workflow errors. A typical workflow has multiple steps. Each step may have a `Status` message for error reporting.
- Batch operations. If a client uses batch request and batch response, the `Status` message should be used directly inside batch response, one for each error sub-response.
- Asynchronous operations. If an API call embeds asynchronous operation results in its response, the status of those operations should be represented directly using the `Status` message.
- Logging. If some API errors are stored in logs, the message `Status` could be used directly after any stripping needed for security/privacy reasons.

Field	Type	Label	Description
code	<a href="#">int32</a>		The status code, which should be an enum value of <a href="#">google.rpc.Code</a> .
message	<a href="#">string</a>		A developer-facing error message, which should be in English. Any user-facing error message should be localized and sent in the <a href="#">google.rpc.Status.details</a> field, or localized by the client.
details	<a href="#">google.protobuf.Any</a>	repeated	A list of messages that carry the error details. There is a common set of message types for APIs to use.

### 3.2.1.4 [google/rpc/error\\_details.proto](#)

#### BadRequest

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

Field	Type	Label	Description
field_violations	<a href="#">BadRequest.FieldViolation</a>	repeated	Describes all violations in a client request.

#### BadRequest.FieldViolation

A message type used to describe a single bad request field.

Field	Type	Label	Description
field	<a href="#">string</a>		A path leading to a field in the request body. The value will be a sequence of dot-separated identifiers that identify a protocol buffer field. E.g., field_violations.field would identify this field.
description	<a href="#">string</a>		A description of why the request element is bad.

#### DebugInfo

Describes additional debugging info.

Field	Type	Label	Description
stack_entries	<a href="#">string</a>	repeated	The stack trace entries indicating where the error occurred.
detail	<a href="#">string</a>		Additional debugging information provided by the server.

#### Help

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Field	Type	Label	Description
links	<a href="#">Help.Link</a>	repeated	URL(s) pointing to additional information on handling the current error.

## Help.Link

Describes a URL link.

Field	Type	Label	Description
description	<a href="#">string</a>		Describes what the link offers.
url	<a href="#">string</a>		The URL of the link.

## LocalizedMessage

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Field	Type	Label	Description
locale	<a href="#">string</a>		The locale used following the specification defined at <a href="http://www.rfc-editor.org/rfc/bcp/bcp47.txt">http://www.rfc-editor.org/rfc/bcp/bcp47.txt</a> . Examples are: en-US, fr-CH, es-MX
message	<a href="#">string</a>		The localized error message in the above locale.

## PreconditionFailure

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

Field	Type	Label	Description
violations	<a href="#">PreconditionFailure.Violation</a>	repeated	Describes all precondition violations.

## PreconditionFailure.Violation

A message type used to describe a single precondition failure.

Field	Type	Label	Description
type	<a href="#">string</a>		The type of PreconditionFailure. We recommend using a service-specific enum type to define the supported precondition violation types. For example, TOS for Terms of Service violation.
subject	<a href="#">string</a>		The subject, relative to the type, that failed. For example, google.com/cloud relative to the TOS type would indicate which terms of service is being referenced.
description	<a href="#">string</a>		A description of how the precondition failed. Developers can use this description to understand how to fix the failure. For example: Terms of service not accepted.

## QuotaFailure

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set `service_disabled` to true.

Also see `RetryDetail` and `Help` types for other details about handling a quota failure.

Field	Type	Label	Description
violations	<a href="#">QuotaFailure.Violation</a>	repeated	Describes all quota violations.

## QuotaFailure.Violation

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Field	Type	Label	Description
subject	<a href="#">string</a>		The subject on which the quota check failed. For example, clientip: or project:
description	<a href="#">string</a>		A description of how the quota check failed. Clients can use this description to find more about the quota configuration in the service's public documentation, or find the relevant quota limit to adjust through developer console. For example: Service disabled or Daily Limit for read operations exceeded.

## RequestInfo

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Field	Type	Label	Description
request_id	<a href="#">string</a>		An opaque string that should only be interpreted by the service generating it. For example, it can be used to identify requests in the service's logs.
serv-ing_data	<a href="#">string</a>		Any data that was used to serve this request. For example, an encrypted stack trace that can be sent back to the service provider for debugging.

## ResourceInfo

Describes the resource that is being accessed.

Field	Type	Label	Description
re-source_type	<a href="#">string</a>		A name for the type of resource being accessed, e.g. sql table, cloud storage bucket, file, Google calendar; or the type URL of the resource: e.g. <a href="#">type.googleapis.com/google.pubsub.v1.Topic</a> .
re-source_name	<a href="#">string</a>		The name of the resource being accessed. For example, a shared calendar name: <a href="#">example.com_4fghdhgsrgh@group.calendar.google.com</a> , if the current error is <a href="#">google.rpc.Code.PERMISSION_DENIED</a> .
owner	<a href="#">string</a>		The owner of the resource (optional). For example, user: or project:.
description	<a href="#">string</a>		Describes what error is encountered when accessing this resource. For example, updating a cloud project may require the writer permission on the developer console project.

## RetryInfo

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until `retry_delay` amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on `retry_delay`, until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

Field	Type	Label	Description
retry_delay	<a href="#">google.protobuf.Duration</a>		Clients should wait at least this long between retrying the same request.

### 3.2.1.5 com/digitalasset/ledger/api/v1/trace\_context.proto

#### TraceContext

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

All identifiers are opaque.

Field	Type	Label	Description
trace_id_high	<code>uint64</code>		If present, this is the high 64 bits of the 128-bit identifier. Otherwise the trace ID is 64 bits long.
trace_id	<code>uint64</code>		The Traceld 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	<code>uint64</code>		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 Traceld.
parent_span_id	<code>google.protobuf.UInt64Value</code>		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	<code>bool</code>		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.

### 3.2.1.6 com/digitalasset/ledger/api/v1/ledger\_identity\_service.proto

#### GetLedgerIdentityRequest

Field	Type	Label	Description
trace_context	<code>TraceContext</code>		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	<code>string</code>		The ID of the ledger exposed by the server. Requests submitted with the wrong ledger ID will result in NOT_FOUND gRPC errors. 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	<code>GetLedgerIdentityRequest</code>	<code>GetLedgerIdentityResponse</code>	Clients may call this RPC to return the identifier of the ledger they are connected to.

### 3.2.1.7 com/digitalasset/ledger/api/v1/ledger\_offset.proto

#### LedgerOffset

Describes a specific point on the ledger.

Field	Type	Label	Description
absolute	<a href="#">string</a>		Absolute values are acquired by reading the transactions in the stream. The offsets can be compared. The format may vary between implementations. It is either: * a string representing an ever-increasing integer * a composite string containing -; ordering requires comparing numerical values of the second, then the third element.
boundary	<a href="#">LedgerOffset.LedgerBoundary</a>		

#### 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.

### 3.2.1.8 com/digitalasset/ledger/api/v1/package\_service.proto

#### GetPackageRequest

Field	Type	Label	Description
ledger_id	<a href="#">string</a>		Must correspond to the ledger id reported by the Ledger Identification Service. Required
package_id	<a href="#">string</a>		The ID of the requested package. Required
trace_context	<a href="#">TraceContext</a>		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	<a href="#">HashFunction</a>		The hash function we use to calculate the hash Required
archive_payload	<a href="#">bytes</a>		Contains a daml_lf ArchivePayload. See further details in daml_lf.proto. Required
hash	<a href="#">string</a>		The hash of the archive payload, can also used as a package_id. Required

## GetPackageStatusRequest

Field	Type	Label	Description
ledger_id	<a href="#">string</a>		Must correspond to the ledger id reported by the Ledger Identification Service. Required
package_id	<a href="#">string</a>		The ID of the requested package. Required
trace_context	<a href="#">TraceContext</a>		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	<a href="#">PackageStatus</a>		The status of the package.

## ListPackagesRequest

Field	Type	Label	Description
ledger_id	<a href="#">string</a>		Must correspond to the ledger id reported by the Ledger Identification Service. Required
trace_context	<a href="#">TraceContext</a>		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	<a href="#">string</a>	repeated	The IDs of all DAML LF packages supported by the server. 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	<a href="#">ListPackagesRequest</a>	<a href="#">ListPackagesResponse</a>	Returns the identifiers of all supported packages.
GetPackage	<a href="#">GetPackageRequest</a>	<a href="#">GetPackageResponse</a>	Returns the contents of a single package, or a NOT_FOUND error if the requested package is unknown.
GetPackageStatus	<a href="#">GetPackageStatusRequest</a>	<a href="#">GetPackageStatusResponse</a>	Returns the status of a single package.

### 3.2.1.9 com/digitalasset/ledger/api/v1/transaction.proto

## Transaction

Filtered view of an on-ledger transaction.

Field	Type	Label	Description
transaction_id	<a href="#">string</a>		Assigned by the server. Useful for correlating logs. Required
command_id	<a href="#">string</a>		The ID of the command which resulted in this transaction. Missing for everyone except the submitting party. Optional
workflow_id	<a href="#">string</a>		The workflow id used in command submission. Optional
effective_at	<a href="#">google.protobuf.Timestamp</a>		Ledger effective time. Required
events	<a href="#">Event</a>	repeated	The collection of events. Only contains CreatedEvent or ArchivedEvent. Required
offset	<a href="#">string</a>		The absolute offset. The format of this field is described in ledger_offset.proto. Required
trace_context	<a href="#">TraceContext</a>		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	<a href="#">string</a>		Assigned by the server. Useful for correlating logs. Required
command_id	<a href="#">string</a>		The ID of the command which resulted in this transaction. Missing for everyone except the submitting party. Optional
workflow_id	<a href="#">string</a>		The workflow id used in command submission. Only set if the workflow_id for the command was set. Optional
effective_at	<a href="#">google.protobuf.Timestamp</a>		Ledger effective time. Required
offset	<a href="#">string</a>		The absolute offset. The format of this field is described in ledger_offset.proto. Required
events_by_id	<a href="#">TransactionTree.EventsByEntry</a>	repeated	Changes to the ledger that were caused by this transaction. Nodes of the transaction tree. Required
root_event_ids	<a href="#">string</a>	repeated	Roots of the transaction tree. Required
trace_context	<a href="#">TraceContext</a>		Zipkin trace context. This field is a future extension point and is currently not supported. Optional

## TransactionTree.EventsByIdEntry

Field	Type	Label	Description
key	<a href="#">string</a>		
value	<a href="#">TreeEvent</a>		

## TreeEvent

Field	Type	Label	Description
created	<a href="#">CreatedEvent</a>		
exercised	<a href="#">ExercisedEvent</a>		

## 3.2.1.10 com/digitalasset/ledger/api/v1/completion.proto

### Completion

A completion represents the status of a submitted command on the ledger, i.e. it can be successful or failed.

Field	Type	Label	Description
com-command_id	<a href="#">string</a>		The ID of the succeeded or failed command. Required
status	<a href="#">google.rpc.Status</a>		Identifies the exact type of the error. Documentation for how command submission errors map to grpc status codes will be provided. e.g. malformed or double spend transactions will result in a INVALID_ARGUMENT status. transactions with invalid time windows but may be valid at a later date will result in an ABORTED error. Optional
trace_context	<a href="#">TraceContext</a>		The trace context submitted with the command. This field is a future extension point and is currently not supported. Optional

## 3.2.1.11 com/digitalasset/ledger/api/v1/transaction\_service.proto

### GetLedgerEndRequest

Field	Type	Label	Description
ledger_id	<a href="#">string</a>		Must correspond to the ledger id reported by the Ledger Identification Service. Required
trace_context	<a href="#">TraceContext</a>		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	<a href="#">LedgerOffset</a>		The absolute offset of the current ledger end.

## GetTransactionByEventIdRequest

Field	Type	Label	Description
ledger_id	<a href="#">string</a>		Must correspond to the ledger id reported by the Ledger Identification Service. Required
event_id	<a href="#">string</a>		The ID of a particular event. Required
requesting_parties	<a href="#">string</a>	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. Required
trace_context	<a href="#">TraceContext</a>		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	<a href="#">string</a>		Must correspond to the ledger id reported by the Ledger Identification Service. Required
transaction_id	<a href="#">string</a>		The ID of a particular transaction. Required
requesting_parties	<a href="#">string</a>	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. Required
trace_context	<a href="#">TraceContext</a>		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	<a href="#">TransactionTree</a>		

## GetTransactionTreesResponse

Field	Type	Label	Description
transactions	<a href="#">Transaction-Tree</a>	repeated	The list of transaction trees that matches the filter in GetTransactionsRequest for the GetTransactionTrees method.

## GetTransactionsRequest

Field	Type	Label	Description
ledger_id	<a href="#">string</a>		Must correspond to the ledger id reported by the Ledger Identification Service. Required
begin	<a href="#">LedgerOffset</a>		Beginning of the requested ledger section. Required
end	<a href="#">LedgerOffset</a>		End of the requested ledger section. Optional, if not set, the stream will not terminate.
filter	<a href="#">Transaction-Filter</a>		Requesting parties with template filters. Required
verbose	<a href="#">bool</a>		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	<a href="#">TraceContext</a>		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	<a href="#">Transaction</a>	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	<a href="#">GetTransactionsRequest</a>	<a href="#">GetTransactionsResponse</a>	Read the ledger's filtered transaction stream for a set of parties.
GetTransactionTrees	<a href="#">GetTransactionsRequest</a>	<a href="#">GetTransactionTreesResponse</a>	Read the ledger's complete transaction stream for a set of parties.
GetTransactionByEventId	<a href="#">GetTransactionByEventIdRequest</a>	<a href="#">GetTransactionResponse</a>	Lookup a transaction by the ID of an event that appears within it. Returns NOT_FOUND if no such transaction exists.
GetTransactionById	<a href="#">GetTransactionByIdRequest</a>	<a href="#">GetTransactionResponse</a>	Lookup a transaction by its ID. Returns NOT_FOUND if no such transaction exists.
GetLedgerEnd	<a href="#">GetLedgerEndRequest</a>	<a href="#">GetLedgerEndResponse</a>	Get the current ledger end. Subscriptions started with the returned offset will serve transactions created after this RPC was called.

### 3.2.1.12 com/digitalasset/ledger/api/v1/value.proto

#### Identifier

Unique identifier of an entity.

Field	Type	Label	Description
package_id	<a href="#">string</a>		The identifier of the DAML package that contains the entity. Required
name	<a href="#">string</a>		The identifier of the entity (unique within the package) DEPRECATED: use module_name and entity_name instead Optional
module_name	<a href="#">string</a>		The dot-separated module name of the identifier. Required
entity_name	<a href="#">string</a>		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	<a href="#">Value</a>	repeated	The elements must all be of the same concrete value type. Optional

## 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		Omitted from the transaction stream when verbose streaming is not enabled. If any of the keys are omitted within a single record, the order of fields MUST match the order of declaration in the DAML template. Optional, when submitting commands.
value	Value		A nested value of a record. Required

## Value

Encodes values that the ledger accepts as command arguments and emits as contract arguments.

Field	Type	Label	Description
record	<a href="#">Record</a>		
variant	<a href="#">Variant</a>		
contract_id	<a href="#">string</a>		Identifier of an on-ledger contract. Commands which reference an unknown or already archived contract ID will fail.
list	<a href="#">List</a>		Represents a homogenous list of values.
int64	<a href="#">sint64</a>		
decimal	<a href="#">string</a>		A decimal value with precision 38 (38 decimal digits), of which 10 after the comma / period. In other words a decimal is a number of the form $x / 10^{10}$ where $ x  < 10^{38}$ . The number can start with a leading sign [+-] followed by digits
text	<a href="#">string</a>		A string.
timestamp	<a href="#">sfixed64</a>		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 <a href="https://www.ietf.org/rfc/rfc3339.txt">https://www.ietf.org/rfc/rfc3339.txt</a>
party	<a href="#">string</a>		An agent operating on the ledger.
bool	<a href="#">bool</a>		True or false.
unit	<a href="#">google.protobuf.Empty</a>		This value is used for example for choices that don't take any arguments.
date	<a href="#">int32</a>		days since the unix epoch. Can go backwards. Limited from 0001-01-01 to 9999-12-31, also to be compatible with <a href="https://www.ietf.org/rfc/rfc3339.txt">https://www.ietf.org/rfc/rfc3339.txt</a>
optional	<a href="#">Optional</a>		The Optional type, None or Some

## Variant

A value with alternative representations.

Field	Type	Label	Description
variant_id	<a href="#">Identifier</a>		Omitted from the transaction stream when verbose streaming is not enabled. Optional when submitting commands.
constructor	<a href="#">string</a>		Determines which of the Variant's alternatives is encoded in this message. Required
value	<a href="#">Value</a>		The value encoded within the Variant. Required

### 3.2.1.13 com/digitalasset/ledger/api/v1/ledger\_configuration\_service.proto

## GetLedgerConfigurationRequest

Field	Type	Label	Description
ledger_id	<a href="#">string</a>		Must correspond to the ledger id reported by the Ledger Identification Service. Required
trace_context	<a href="#">TraceContext</a>		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	<a href="#">LedgerConfiguration</a>		The latest ledger configuration.

## LedgerConfiguration

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

Field	Type	Label	Description
min_ttl	<a href="#">google.protobuf.Duration</a>		Minimum difference between ledger effective time and maximum record time in submitted commands.
max_ttl	<a href="#">google.protobuf.Duration</a>		Maximum difference between ledger effective time and maximum record time in submitted commands.

## LedgerConfigurationService

LedgerConfigurationService allows clients to subscribe to changes of the ledger configuration.

Method Name	Request Type	Response Type	Description
GetLedger-Configura-tion	<a href="#">GetLedger-Configura-tionRequest</a>	<a href="#">GetLedger-Configura-tionResponse</a>	GetLedgerConfiguration returns the latest configura-tion as the first response, and publishes configura-tion updates in the same stream.

## 3.2.1.14 com/digitalasset/ledger/api/v1/testing/reset\_service.proto

### ResetRequest

Field	Type	Label	Description
ledger_id	<a href="#">string</a>		Must correspond to the ledger id reported by the Ledger Identification Service. 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	<a href="#">ResetRequest</a>	<a href="#">.google.protobuf.Empty</a>	Resets the ledger state. Note that loaded DARs won't be removed - this only rolls back the ledger to genesis.

### 3.2.1.15 com/digitalasset/ledger/api/v1/testing/time\_service.proto

#### GetTimeRequest

Field	Type	Label	Description
ledger_id	<a href="#">string</a>		Must correspond to the ledger id reported by the Ledger Identification Service. Required

#### GetTimeResponse

Field	Type	Label	Description
current_time	<a href="#">google.protobuf.Timestamp</a>		The current time according to the ledger server.

## SetTimeRequest

Field	Type	Label	Description
ledger_id	<code>string</code>		Must correspond to the ledger id reported by the Ledger Identification Service. Required
current_time	<code>google.protobuf.Timestamp</code>		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	<code>google.protobuf.Timestamp</code>		The time the client wants to set on the ledger. MUST be a point in time after current_time.

## TimeService

Optional service, exposed for testing static time scenarios.

Method Name	Request Type	Response Type	Description
GetTime	<code>GetTimeRequest</code>	<code>GetTimeResponse</code>	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	<code>SetTimeRequest</code>	<code>.google.protobuf.Empty</code>	Allows clients to change the ledger's clock in an atomic get-and-set operation.

### 3.2.1.16 com/digitalasset/ledger/api/v1/event.proto

#### ArchivedEvent

Records that a contract has been archived, and choices may no longer be exercised on it.

Field	Type	Label	Description
event_id	<code>string</code>		The ID of this particular event. Required
contract_id	<code>string</code>		The ID of the archived contract. Required
template_id	<code>Identifier</code>		The template of the archived contract. Required
witness_parties	<code>string</code>	repeated	The parties that are notified of this event. Required

#### CreatedEvent

Records that a contract has been created, and choices may now be exercised on it.

Field	Type	Label	Description
event_id	<a href="#">string</a>		The ID of this particular event. Required
contract_id	<a href="#">string</a>		The ID of the created contract. Required
template_id	<a href="#">Identifier</a>		The template of the created contract. Required
create_arguments	<a href="#">Record</a>		The arguments that have been used to create the contract. Required
witness_parties	<a href="#">string</a>	repeated	The parties that are notified of this event. Required

## Event

An event on the ledger can either be the creation or the archiving of a contract, or the exercise of a choice on a contract. The GetTransactionTrees response will only contain create and exercise events. Archive events correspond to consuming exercise events.

Field	Type	Label	Description
created	<a href="#">CreatedEvent</a>		
exercised	<a href="#">ExercisedEvent</a>		
archived	<a href="#">ArchivedEvent</a>		

## 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. Required
contract_id	string		The ID of the target contract. Required
template_id	Identifier		The template of the target contract. Required
contract_creating_event_id	string		The ID of the event in which the target contract has been created. Required
choice	string		The choice that's been exercised on the target contract. Required
choice_argument	Value		The argument the choice was made with. Required
acting_parties	string	repeated	The parties that made the choice. 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. 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. Optional

### 3.2.1.17 com/digitalasset/ledger/api/v1/command\_completion\_service.proto

#### Checkpoint

Checkpoints may be used to:

- \* detect time out of commands.
- \* provide an offset which can be used to restart consumption.

Field	Type	Label	Description
record_time	google.protobuf.Timestamp		All commands with a maximum record time below this value MUST be considered lost if their completion has not arrived before this checkpoint. Required
offset	LedgerOffset		May be used in a subsequent CompletionStreamRequest to resume the consumption of this stream at a later time. Required

## CompletionEndRequest

Field	Type	Label	Description
ledger_id	<a href="#">string</a>		Must correspond to the ledger id reported by the Ledger Identification Service. Required
trace_context	<a href="#">TraceContext</a>		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	<a href="#">LedgerOffset</a>		This offset can be used in a CompletionStreamRequest message. Required

## CompletionStreamRequest

Field	Type	Label	Description
ledger_id	<a href="#">string</a>		Must correspond to the ledger id reported by the Ledger Identification Service. Required
application_id	<a href="#">string</a>		Only completions of commands submitted with the same application_id will be visible in the stream. Required
parties	<a href="#">string</a>	repeated	Non-empty list of parties whose data should be included. Required
offset	<a href="#">LedgerOffset</a>		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	<a href="#">Checkpoint</a>		This checkpoint may be used to restart consumption. The checkpoint is after any completions in this response. Optional
completions	<a href="#">Completion</a>	repeated	If set, one or more completions.

## CommandCompletionService

Allows clients to observe the status of their submissions. Commands may be submitted via the Command Submission Service. The on-ledger effects of their submissions are disclosed by the Transac-

tion Service. Commands may fail in 4 distinct manners: 1) INVALID\_PARAMETER gRPC error on malformed payloads and missing required fields. 2) Failure communicated in the gRPC error. 3) Failure communicated in a Completion. 4) A Checkpoint with record\_time > command mrt arrives through the Completion Stream, and the command's Completion was not visible before. In this case the command is lost. Clients that do not receive a successful completion about their submission MUST NOT assume that it was successful. Clients SHOULD subscribe to the CompletionStream before starting to submit commands to prevent race conditions.

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

Method Name	Request Type	Response Type	Description
Completion-Stream	<a href="#">CompletionStream-Request</a>	<a href="#">CompletionStreamResponse</a>	Subscribe to command completion events.
CompletionEnd	<a href="#">CompletionEndRequest</a>	<a href="#">CompletionEndResponse</a>	Returns the offset after the latest completion.

### 3.2.1.18 [com/digitalasset/ledger/api/v1/command\\_service.proto](#)

#### SubmitAndWaitRequest

These commands are atomic, and will become transactions.

Field	Type	Label	Description
commands	<a href="#">Commands</a>		The commands to be submitted. Required
trace_context	<a href="#">TraceContext</a>		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	<a href="#">SubmitAndWaitRequest</a>	<a href="#">.google.protobuf.buf.Empty</a>	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.

### 3.2.1.19 com/digitalasset/ledger/api/v1/transaction\_filter.proto

#### Filters

Field	Type	Label	Description
inclusive	<a href="#">InclusiveFilters</a>		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
tem-plate_ids	<a href="#">Identifier</a>	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
fil-ters_by_party	<a href="#">Transaction-Filter.Filters-ByPartyEntry</a>	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. Required

#### TransactionFilter.FiltersByPartyEntry

Field	Type	Label	Description
key	<a href="#">string</a>		
value	<a href="#">Filters</a>		

### 3.2.1.20 com/digitalasset/ledger/api/v1/commands.proto

#### Command

A command can either create a new contract or exercise a choice on an existing contract.

Field	Type	Label	Description
create	<a href="#">CreateCommand</a>		
exercise	<a href="#">ExerciseCommand</a>		

## Commands

A composite command that groups multiple commands together.

Field	Type	Label	Description
ledger_id	<a href="#">string</a>		Must correspond to the ledger id reported by the Ledger Identification Service. Required
work-flow_id	<a href="#">string</a>		Identifier of the on-ledger workflow that this command is a part of. Optional
application_id	<a href="#">string</a>		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. Required
command_id	<a href="#">string</a>		Unique command id. This number should be unique for each new command within an application domain. It can be used for matching the requests with their respective completions. Required
party	<a href="#">string</a>		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. Required
ledger_effective_time	<a href="#">google.protobuf.Timestamp</a>		MUST be an approximation of the wall clock time on the ledger server. Required
maximum_record_time	<a href="#">google.protobuf.Timestamp</a>		The deadline for observing this command in the completion stream before it can be considered to have timed out. Required
commands	<a href="#">Command</a>	repeated	Individual elements of this atomic command. Must be non-empty. Required

## CreateCommand

Create a new contract instance based on a template.

Field	Type	Label	Description
template_id	<a href="#">Identifier</a>		The template of contract the client wants to create. Required
create_arguments	<a href="#">Record</a>		The arguments required for creating a contract from this template. Required

## ExerciseCommand

Exercise a choice on an existing contract.

Field	Type	Label	Description
template_id	<a href="#">Identifier</a>		The template of contract the client wants to exercise. Required
contract_id	<a href="#">string</a>		The ID of the contract the client wants to exercise upon. Required
choice	<a href="#">string</a>		The name of the choice the client wants to exercise. Required
choice_argument	<a href="#">Value</a>		The argument for this choice. Required

### [3.2.1.21 com/digitalasset/ledger/api/v1/command\\_submission\\_service.proto](#)

## SubmitRequest

The submitted commands will be processed atomically in a single transaction.

Field	Type	Label	Description
commands	<a href="#">Commands</a>		The commands to be submitted in a single transaction. Required
trace_context	<a href="#">TraceContext</a>		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	<a href="#">SubmitRequest</a>	<a href="#">.google.protobuf.Empty</a>	Submit a single composite command.

### 3.2.1.22 com/digitalasset/ledger/api/v1/active\_contracts\_service.proto

#### GetActiveContractsRequest

Field	Type	Label	Description
ledger_id	<a href="#">string</a>		Must correspond to the ledger id reported by the Ledger Identification Service. Required
filter	<a href="#">Transaction-Filter</a>		Templates to include in the served snapshot, per party. Required
verbose	<a href="#">bool</a>		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	<a href="#">TraceContext</a>		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	<a href="#">string</a>		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	<a href="#">string</a>		The workflow that created the contracts. Optional
active_contracts	<a href="#">CreatedEvent</a>	repeated	The list of contracts that were introduced by the workflow with workflow_id at the offset. Optional
trace_context	<a href="#">TraceContext</a>		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
GetActive-Contracts	<a href="#">GetActive-ContractsRequest</a>	<a href="#">GetActive-ContractsResponse</a>	Returns a stream of the latest snapshot of active contracts. Getting an empty stream means that the active contracts set is empty and the client should listen to transactions using LEDGER_BEGIN Clients SHOULD NOT assume that the set of active contracts they receive reflects the state at the ledger end.

### 3.2.1.23 Scalar Value Types

.proto Type	Notes	C++ Type	Java Type	Python Type
double		double	double	float
float		float	float	float
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int32	int	int
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	int64	long	int/long
uint32	Uses variable-length encoding.	uint32	int	int/long
uint64	Uses variable-length encoding.	uint64	long	int/long
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32	int	int
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	int64	long	int/long
fixed32	Always four bytes. More efficient than uint32 if values are often greater than $2^{28}$ .	uint32	int	int
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than $2^{56}$ .	uint64	long	int/long
sfixed32	Always four bytes.	int32	int	int
sfixed64	Always eight bytes.	int64	long	int/long
bool		bool	boolean	boolean
string	A string must always contain UTF-8 encoded or 7-bit ASCII text.	string	String	str/unicode
bytes	May contain any arbitrary sequence of bytes.	string	ByteString	str

### 3.2.2 From DAML to Ledger API

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.

## On this page

- [From DAML to Ledger API](#)
- [Notation](#)
- [Records and Primitive Types](#)
- [Variants](#)
- [Contract Templates](#)
  - \* [Creating a Contract](#)
  - \* [Receiving a Contract](#)
  - \* [Exercising a Choice](#)

### 3.2.2.1 Notation

The notation used on this page for the protobuf messages is the same as you get if you invoke `protoc --decode=Foo < some_payload.bin`. To illustrate the notation, here is a simple definition of the messages Foo and Bar:

```
// Copyright (c) 2019 Digital Asset (Switzerland) GmbH and/or its
// affiliates. All rights reserved.
// SPDX-License-Identifier: Apache-2.0

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.

### 3.2.2.2 Records and Primitive Types

Records or product types are translated to [Record](#). Here's an example DAML record type that contains a field for each primitive type:

```
data MyProductType = MyProductType {
    intField: Int;
    textField: Text;
    decimalField: Decimal;
    boolField: Bool;
    partyField: Party;
    timeField: Time;
    listField: List Int;
    contractIdField: ContractId SomeTemplate
}
```

And here's an example of creating a value of type `MyProductType`:

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

(continues on next page)

(continued from previous page)

```

    }
}

fields { // RecordField
  label: "boolField"
  value { // Value
    bool: false
  }
}
fields { // RecordField
  label: "partyField"
  value { // Value
    party: "Bob"
  }
}
fields { // RecordField
  label: "timeField"
  value { // Value
    timestamp: 1526428800000000
  }
}
fields { // RecordField
  label: "listField"
  value { // Value
    list { // List
      elements { // Value
        int64: 1
      }
      elements { // Value
        int64: 2
      }
      elements { // Value
        int64: 3
      }
    }
  }
}
fields { // RecordField
  label: "contractIdField"
  value { // Value
    contract_id: "some-contract-id"
  }
}
}

```

### 3.2.2.3 Variants

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

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

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

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

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

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

Listing 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
          }
        }
      }
    }
  }
}
```

(continues on next page)

(continued from previous page)

```

        }
    }
}
}
}
```

### 3.2.2.4 Contract Templates

Contract templates are represented as records with the same identifier as the template.

This first example template below contains only the signatory party and a simple choice to exercise:

```

template MySimpleTemplate
  with
    owner: Party
  where
    signatory owner

  controller owner can
    MyChoice
      : ()
      with parameter: Int
      do return ()
```

### Creating a Contract

Creating contracts is done by sending a [CreateCommand](#) to the [CommandSubmissionService](#) or the [CommandService](#). The message to create a `MySimpleTemplate` contract with `Alice` being the owner is shown below:

```
{
  // CreateCommand
  template_id { // Identifier
    package_id: "some-hash"
    name: "Templates.MySimpleTemplate"
  }
  create_arguments { // Record
    fields { // RecordField
      label: "owner"
      value { // Value
        party: "Alice"
      }
    }
  }
}
```

## Receiving a Contract

Contracts are received from the [TransactionService](#) in the form of a [CreatedEvent](#). The data contained in the event corresponds to the data that was used to create the contract.

```
{ // CreatedEvent
  event_id: "some-event-id"
  contract_id: "some-contract-id"
  template_id { // Identifier
    package_id: "some-hash"
    name: "Templates.MySimpleTemplate"
  }
  create_arguments { // Record
    fields { // RecordField
      label: "owner"
      value { // Value
        party: "Alice"
      }
    }
  }
  witness_parties: "Alice"
}
```

## Exercising a Choice

A choice is exercised by sending an [ExerciseCommand](#). Taking the same contract template again, exercising the choice `MyChoice` would result in a command similar to the following:

```
{ // ExerciseCommand
  template_id { // Identifier
    package_id: "some-hash"
    name: "Templates.MySimpleTemplate"
  }
  contract_id: "some-contract-id"
  choice: "MyChoice"
  choice_argument { // Value
    record { // Record
      fields { // RecordField
        label: "parameter"
        value { // Value
          int64: 42
        }
      }
    }
  }
}
```

### 3.2.3 gRPC

gRPC is an RPC framework that uses Google Protocol Buffers to define services. A service is comprised of methods with a request message and a response message. Requests and responses can also be streams.

The reasons for adopting gRPC as the basis for the Ledger API were manifold:

- It is an open-source interoperability mechanism with multitude of language mappings
- It does not tie the users to a specific technology stack
- It is backed up by the giants of the industry with vast adoption examples and vivid community of contributors
- It uses binary representation for wire transfer which has been designed for scalability and high performance
- It has been designed with simplicity in mind, a departure from the likes of CORBA, DCOM, RMI and other RPCs
- It enables modelling flexibility beyond the rigid set of verbs offered by HTTP and its request-response architecture
- It supports both uni- as well as bidirectional streaming
- It is built on top of HTTP2 and thus understood by off-the-shelf infrastructure components such as firewalls, load balancers, proxies etc.
- It provides secure connections in the form of SSL/TLS as well as pluggable authentication mechanisms
- It simplifies SDK deployment. In order to be productive, clients just need to compile source files to generate the stubs for their target language and copy sample code (if provided)

For more information consult the [gRPC website](#).

### 3.2.4 Getting started

The complete specification in form of protobuf files can be added to an existing SDK project by running `da project add ledger-api-protos`. While gRPC is available for a wide range of programming languages, the DAML SDK provides pre-compiled client bindings for the Ledger API for Java. If you are using a different programming language, please see the [Quickstart](#) and [Tutorials](#) section in the official gRPC documentation.

#### 3.2.4.1 Example project - Ping Pong

The SDK provides an example project demonstrating the use of the Ledger API. Read [3. Configure Maven](#) to configure your machine to use the examples.

1. Start a new project with `da new example-ping-pong-grpc-java ping-pong-java`. This creates a new SDK project in the folder `./ping-pong-java` that is comprised of a DAML model in `daml/PingPong.daml` and Java source code.
2. Run `mvn compile` to build the Java code with [Maven](#).
3. Run `da sandbox` to start the sandbox.
4. Run `mvn exec:java` to run the application.

The DAML model contains two contract templates `Ping` and `Pong` to showcase how two parties can interact via the ledger. Party `Alice` creates a `Ping` contract with Party `Bob` as the receiver, who will exercise the `RespondPong` choice on that contract. The execution of the choice results in a `Pong` contract being created with `Alice` as the receiver, who can choose to exercise the `RespondPing`

choice on that contract. This creates another Ping contract, effectively starting a new ping pong cycle, until this has happened a number of times, as defined within the DAML model.

The entry point for the Java code is the main class `src/main/java/examples/pingpong/PingPongMain.java`. It shows how to connect to and interact with the DML Ledger via the gRPC API. The code is augmented with comments that describe the various parts of the code. Each party will connect to the ledger, subscribe to the transaction stream, and send initial Ping contracts. The application prints statements similar to the ones shown below.

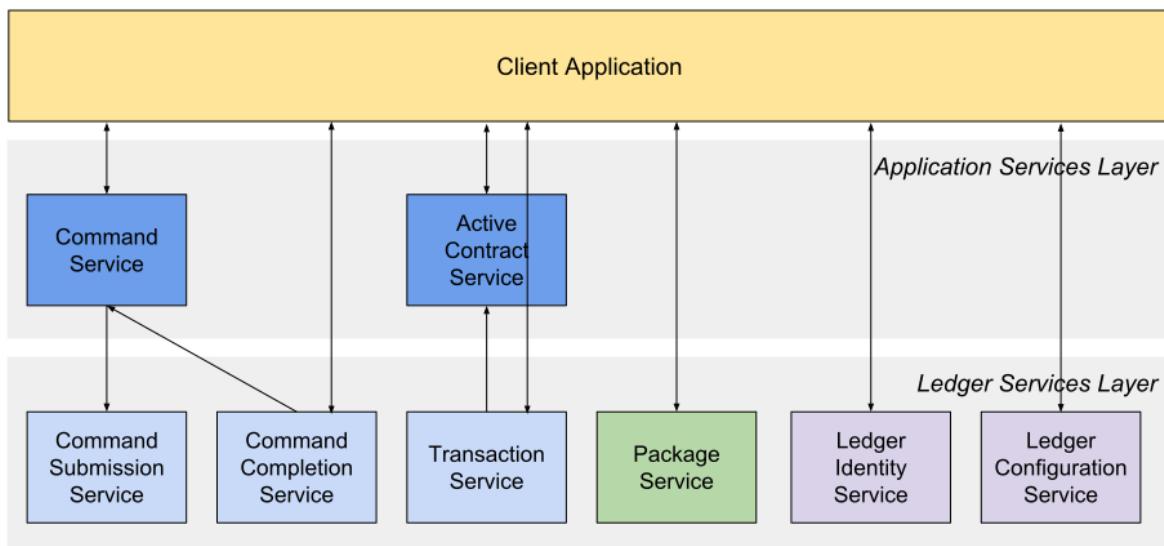
```
Bob is exercising RespondPong on #1:0 in workflow Ping-Alice-1 at count 0
Alice is exercising RespondPing on #344:1 in workflow Ping-Alice-7 at count
 ↵9
```

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.

Note that the Ping Pong 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.

### 3.2.5 Service layers

The Ledger API defines two layers of services. Ledger Services make up the core functionality that is needed to interact with the ledger, while the Application Services provide higher level components that aid developers in building robust applications. See the following chart for an overview:



Consult the [Ledger API Reference](#) for more information on the services and their respective RPC methods.

### 3.2.5.1 Ledger services

Services defined in this layer provide the basic functionality to work with the ledger.

<b>Ledger Administration</b>	
Ledger Identity Service	To retrieve the Ledger ID of the ledger the application is connected to.
Ledger Configuration Service	To retrieve some dynamic properties of the ledger, like minimum and maximum TTL for commands.
<b>Ledger Metadata Exchange</b>	
Package Service	To query the DAML packages deployed to the ledger.
<b>Ledger Data Exchange</b>	
Command Submission Service	To submit commands to the ledger.
Command Completion Service	To track the status of submitted commands.
Transaction Service	To retrieve transactions of events from the ledger.

### 3.2.5.2 Application services

These services provide high level functionality that is common to many applications.

<b>Ledger Data Exchange</b>	
Command Service	Combines command submission and command completion into a single service.
Active Contract Service	To quickly bootstrap an application with active contracts. It eliminates the necessity to read from the beginning of the ledger and to process create events for contracts that have already been archived.

### 3.2.6 Transaction and transaction trees

TransactionService offers several different subscriptions. The most commonly used is the GetTransactions service demonstrated in the example above. 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.

### 3.2.7 DAML-LF

DAML templates, statements and expressions are compiled into a machine readable format called DAML-Ledger Fragment, abbreviated as DAML-LF. It is a canonical representation interpreted by the DAML engine. A good analogy for the relationship between the surface DAML syntax and DAML-LF format is that between Java and JVM bytecode.

DAML-LF content appears in the package service interactions. It is represented as opaque blobs that require a secondary decoding phase. Doing it this way allows employing separate versioning pace for DAML-LF and also opens up the possibility for multiple DAML-LF versions to be supported by a single Ledger API protocol version.

### 3.2.8 Commonly used types

Ledger API uses a number of identifier fields that are represented as strings in the .proto files. They are opaque and shouldn't be interpreted by the client code. This includes

- Transaction ids
- Event ids
- Contract ids
- Package ids (part of template identifiers)

Complementary to that, there are identifiers that are determined by the client code that won't be interpreted by the server and will be transparently passed to the responses. This includes

- Command ids** used to uniquely identify a command and to match it against its response
- Application id** used to uniquely identify client process talking to the server. A combination of command-id and application-id might be used for deduplication
- Workflow id** used to identify chains of transactions. This can be used to correlate transactions sent across longer time span and by different parties.

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

### 3.2.9 Error handling

See the [gRPC documentation](#) for the standard error codes that the server or the client might return.

For submitted commands, the client should interpret response codes in the following way:

**ABORTED** The platform failed to record the result of the command due to a transient server side error or a time constraint violation. The client is free to retry the submission with updated Ledger Effective Time (LET) and Maximum Record Time (MRT) values.

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

**OK, INTERNAL, UNKNOWN (when returned by the Command Submission Service)** Client should 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)** Client can be sure that the command was successful.

**INTERNAL, UNKNOWN (when returned by the Command Service)** Client should resubmit the command with the same command\_id.

### 3.2.10 Verbosity

The API works in a non-verbose mode by default, which means that some identifiers are omitted:

- Record Ids
- Record Field Labels
- Variant Ids

You can explicitly enable verbosity in requests related to Transactions by setting the verbose field in message GetTransactionsRequest or GetActiveContractsRequest to true, which will result in responses containing the above mentioned identifiers.

### 3.2.11 Limitations

At the moment the platform does not impose any limits on the communication over the Ledger API and all requests are serviced on a best effort basis. It is very likely to change in a short perspective and will involve one or more of the following

- Maximum size of inbound gRPC messages (defined globally, per-service, or per-rpc).
- Maximum number of commands packed together in a composite request.
- Maximum number of resulting events in a transaction.

### 3.2.12 Versioning

On the specification level, Ledger API uses standard protobuf versioning approach. At the moment, all the definitions are grouped into a version 1 (v1) namespace. Minor changes involving additions and removals of the fields are backwards compatible and work seamlessly across clients and servers running different variants of the protocol. Major future alterations that will involve reshaping the constituent records, their nesting, and modifications of the field types will amount to a breaking change and will require issuing a completely new protocol version, separated into a new namespace, likely v2.

On the deployment level, software packages containing Ledger API are versioned using [semantic versioning](#). This enables first-glance discovery of any alterations compared to previous versions.

### 3.2.13 Authentication

Connecting to the The DAML PaaS requires client authentication using certificates. Please refer to the DAML PaaS documentation for further information on this topic. Client authentication is not required for the sandbox.

## 3.3 Java Binding

### 3.3.1 Generating Java code from DAML

#### 3.3.1.1 Introduction

When writing applications for the ledger in Java, you want to work with a representation of DAML templates and data types in Java that closely resemble the original DAML code while still being as true to the native types in Java as possible. To achieve this, you can use DAML to Java code generator (Java codegen) to generate Java types based on a DAML model. You can then use these types in your Java code when reading information from and sending data to the ledger.

#### 3.3.1.2 Downloading

You can download the [latest version](#) of the Java codegen.

### 3.3.1.3 Running 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 archives](#). To use the Java codegen, run this command in a terminal:

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

Use this command to display the helptext:

## Generating 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
    ^^^^^^^^^^
```

## Generating 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 `fromIdAndRecord` 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
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

## Integration 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 [Running the Java codegen](#) just like any other external process (for example the

protobuf compiler). Alternatively you can integrate it as a runnable dependency in your `pom.xml` file for Maven.

The following snippet is an excerpt from the `pom.xml` that is part of the [Quickstart guide](#) guide.

```

<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
    <version>1.6.0</version>
    <dependencies>
        <dependency>
            <groupId>com.daml.java</groupId>
            <artifactId>codegen</artifactId>
            <version>__VERSION__</version>
            <type>jar</type>
        </dependency>
    </dependencies>
    <executions>
        <execution>
            <id>daml-codegen-java</id>
            <phase>generate-sources</phase>
            <goals>
                <goal>java</goal>
            </goals>
            <configuration>
                <includeProjectDependencies>false</
            <!--includeProjectDependencies>
                <includePluginDependencies>true</
            <!--includePluginDependencies>
                <mainClass>com.digitalasset.daml.lf.codegen.Main</
            <!--mainClass>
                <arguments>
                    <argument>-o</argument>
                    <argument>${daml-codegen-java.output}</
            <!--argument>
                <argument>-d</argument>
                <argument>com.digitalasset.quickstart.iou.
            <!--TemplateDecoder</argument>
                <argument>${project.build.directory}/daml/iou.
            <!--dar=com.digitalasset.quickstart.model</argument>
                </arguments>
                <sourceRoot>${daml-codegen-java.output}</
            <!--sourceRoot>
                </configuration>
            </execution>
        </executions>
    </plugin>

```

### 3.3.1.4 Generated Java model

The Java codegen generates Java source files in a directory tree under the output directory specified on the command line.

#### How DAML built-in types map to Java types

DAML built-in types are translated to the following equivalent types in Java:

DAML type	Java type
Int	java.lang.Long
Decimal	java.math.BigDecimal
Text	java.lang.String
Bool	java.util.Boolean
Party	java.lang.String
Date	java.time.LocalDate
Time	java.time.Instant
List or []	java.util.List
Optional	java.util.Optional
() (Unit)	Since Java doesn't have an equivalent of DAML's Unit type () in the standard library, the generated code uses <a href="#">com.daml.ledger.javaapi.data.Unit</a> from the Java Bindings library.
ContractId	Fields of type ContractId X refer to the generated ContractId class of the respective template X.

#### Generated class details

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 or 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.daml

```
daml 1.2
module Com.Acme where

data Person = Person with name : Name; age : Decimal
data Name = Name with firstName : Text; lastName : Text
```

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

Listing 6: com/acme/Person.java

```
package com.acme;

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

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

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

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

Listing 7: com/acme/Name.java

```
package com.acme;

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

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

    public Name(String fistName, String lastName) { /* ... */ }
    public Record toValue() { /* ... */ }
}
```

## Templates

The Java codegen generates three classes for a DAML template:

**TemplateName** Represents the contract data or the template fields.

**TemplateName.ContractId** Used whenever a contract ID of the corresponding template is used in another template or record, for example: data Foo = Foo (ContractId Bar). This class also provides methods to generate an ExerciseCommand for each choice that can be sent to the ledger with the Java Bindings.. TODO: refer to another section explaining exactly that, when we have it.

**TemplateName.Contract** Represents an actual contract on the ledger. It contains a field for the contract ID (of type `TemplateName.ContractId`) and a field for the template data (of type `TemplateName`). With the static method `TemplateName.Contract.fromIdAndRecord`, you can deserialize a `CreatedEvent` to an instance of `TemplateName.Contract`.

Listing 8: Com/Acme.daml

```
daml 1.2
module Com.Acme where

template Bar
  with
    owner: Party
    name: Text

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

A file Java file is generated that defines three Java classes:

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

Listing 9: com/acme/Bar.java

```
package com.acme;

public class Bar extends Template {

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

  public final String owner;
  public final String name;

  public static class ContractId {
    public final String contractId;

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

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

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

(continues on next page)

(continued from previous page)

```

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

    public static Contract fromIdAndRecord(String contractId, Record
    ↪record) { /* ... */ }
}
}

```

## Variants or 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.daml

```

daml 1.2
module Com.Acme where

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

```

The Java code generated for this variant is:

Listing 11: com/acme/BookAttribute.java

```

package com.acme;

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

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

```

Listing 12: com/acme/bookattribute/Pages.java

```

package com.acme.bookattribute;

public class Pages extends BookAttribute {
    public final Long longValue;

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

    public Pages(Long longValue) { /* ... */ }
}

```

(continues on next page)

(continued from previous page)

```
public Value toValue() { /* ... */ }
```

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

```
package com.acme.bookattribute;

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

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

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

}
```

Listing 14: com/acme/bookattribute&gt;Title.java

```
package com.acme.bookattribute;

public class Title extends BookAttribute {
    public final String stringValue;

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

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

}
```

Listing 15: com/acme/bookattribute/Published.java

```
package com.acme.bookattribute;

public class Published extends BookAttribute {
    public final Long year;
    public final String publisher;

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

    public Published(Long year, String publisher) { /* ... */ }
    public Record toValue() { /* ... */ }

}
```

The Java Binding is a client implementation of the Ledger API [based on RxJava](#). It provides an idiomatic way to write DAML Ledger applications.

#### See also:

This documentation for the Java Binding API includes the [JavaDoc reference documentation](#).

### 3.3.2 Overview

The Java Binding library is composed of:

**The Data Layer** A Java-idiomatic layer based on the Ledger API generated classes. This layer simplifies the code required to work with the Ledger API.

Can be found in the java package `com.daml.ledger.javaapi.data`.

**The Reactive Layer** A thin layer built on top of the Ledger API services generated classes.

For each Ledger API service, there is a reactive counterpart with a matching name. For instance, the reactive counterpart of `ActiveContractsServiceGrpc` is `ActiveContractsClient`.

The Reactive Layer also exposes the main interface representing a client connecting via the Ledger API. This interface is called `LedgerClient` and the main implementation working against the DAML Ledger is the `DamlLedgerClient`.

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

**The Reactive Components** A set of optional components you can use to assemble DAML Ledger applications.

The most important components are:

- the `LedgerView`, which provides a local view of the Ledger
- the `Bot`, which provides utility methods to assemble automation logic for the Ledger

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

#### 3.3.2.1 LedgerClient

Connections to the ledger are made by creating instance of classes that implement the interface `LedgerClient`. The class `DamlLedgerClient` implements this interface, and is used to connect to a DA ledger.

This class provides access to the `ledgerId`, and all clients that give access to the various ledger services, such as the active contract set, the transaction service, the time service, etc. This is described [below](#). Consult the [JavaDoc for DamlLedgerClient](#) for full details.

#### 3.3.2.2 LedgerView

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

---

##### Note:

A contract is active if it exists in the Ledger and has not yet been archived.

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

The `LedgerView` is updated every time:

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

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

### 3.3.2.3 Bot

The Bot is an abstraction used to write automation for the DAML Ledger. It is conceptually defined by two aspects:

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

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

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

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

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

```
Bot.wire(String applicationId,
          LedgerClient ledgerClient,
          TransactionFilter transactionFilter,
          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.

## 3.3.3 Getting started

The Java Binding library can be added to a [Maven](#) project. Read [3. Configure Maven](#) to configure your machine.

### 3.3.3.1 Setup a Maven project

To use the Java Binding library, please add to your Maven project the dependency:

```
<dependency>
  <groupId>com.daml.ledger</groupId>
  <artifactId>bindings-rxjava</artifactId>
```

(continues on next page)

(continued from previous page)

```
<version>x.y.z</version>
</dependency>
```

Replace `x.y.z` with the version that you want to use. You can find the available versions at <https://digitalassetsdk.bintray.com/DigitalAssetSDK/com/daml/ledger/bindings-rxjava/>.

### 3.3.3.2 Connecting to the ledger

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

### 3.3.3.3 Connecting securely

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

Secure connections to a DAML Ledger must be configured to use client authentication certificates, which can be provided by a Ledger Operator.

For information on how to set up an `SSLContext` with the provided certificates for client authentication, please consult the gRPC documentation on [TLS with OpenSSL](#) as well as the [HelloWorldClientTls](#) example of the `grpc-java` project.

### 3.3.3.4 Advanced connection settings

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

### 3.3.3.5 Example

To try out the Java Binding library, use one of the two example projects.

There's one with and one without the Reactive Components. 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 is the following:

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 1 and 2 are repeated until the maximum number of contracts defined in the DAML is reached.

### 3.3.3.6 Setting up the example projects

To set up the examples project, clone the public GitHub repository at: <https://github.com/digital-asset/ex-java-bindings> and follow the setup instruction in the README file. This project contains three examples of the PingPong application, built with gRPC (non-Reactive), Reactive and Reactive Component bindings respectively.

### 3.3.3.7 Example project – Ping Pong without reactive components

#### PingPongMain.java

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

how to connect to and interact with the DML Ledger via the Java Binding library  
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.

### 3.3.3.8 Example project – Ping Pong with reactive components

#### PingPongMain.java

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

how to connect to and interact with the DML Ledger via the Java Binding library  
how to use the Reactive Components to build an automation for both parties

#### PingPongBot

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

The `PingPongBot` has two important methods:

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

#### Output

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

### 3.3.4 The underlying library: RxJava

The Java Binding is [RxJava](#), a library for composing asynchronous and event-based programs using observable sequences for the Java VM. It is part of the family of libraries called [ReactiveX](#).

ReactiveX was chosen as the underlying library for the Java Binding because many services that the DAML Ledger offers are exposed as streams of events. So an application that wants to interact with the DAML Ledger must react to one or more DAML Ledger streams.

## 3.4 How are DAML types translated to DAML-LF?

This page explains what DAML-LF is, and 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.

### 3.4.1 What is 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.

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)

### 3.4.2 When you need to know about DAML-LF

Knowledge of DAML-LF can be helpful when using the Ledger API or bindings on top of it. Development is easier if you know what the types in your DAML code look like at the DAML-LF level.

For example, if you are writing an application in Java that creates some DAML contracts, you need to construct values to pass as parameters to the contract. These values are determined by the Java classes generated from DAML-LF - specifically, by the DAML-LF types in that contract template. This means you need an idea of how the DAML-LF types correspond to the types in the original DAML model.

For the most part the translation of types from DAML to DAML-LF should not be surprising. This page goes through all the cases in detail.

For the bindings to your specific programming language, you should refer to the language-specific documentation.

### 3.4.3 Translation of DAML types to DAML-LF

#### 3.4.3.1 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`.

### 3.4.3.2 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 `ghc-prim:GHC.Tuple:TupleN`, for each particular `N` (where  $2 \leq N \leq 20$ ). 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 `ghc-prim:GHC.Tuple:Tuple2 Int64 Text`.

### 3.4.3.3 Data types

DAML-LF has two kinds of data declarations:

**Record** types, which define a collection of data

**Variant** or **sum** types, which define a number of alternatives

[Data type declarations in DAML](#) (starting with the `data` keyword) are translated to either record or variant 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 { fool: Int; foo2: Text }	record Foo { fool: 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 = Bar Int   Baz ()	variant Foo Bar Int64   Baz Unit
data Foo = Bar Int   Baz	variant Foo Bar Int64   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 }	variant Foo Bar Foo.Bar   Baz Text, record Foo.Bar { bar: Int64 }
data Foo = Foo { foo: Int }	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 }

## 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 }

### 3.4.3.4 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 }
```

### 3.4.3.5 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
```

(continues on next page)

(continued from previous page)

```
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 }
```

# Chapter 4

## SDK Tools

### 4.1 SDK Assistant

The SDK Assistant is a command-line tool designed to help you interact with the DAML SDK. It includes commands to help you create projects, run other SDK tools, install and upgrade SDK releases, and view documentation.

#### 4.1.1 Installing the SDK Assistant

Refer to [Installing the SDK](#) for instructions on how to install the SDK Assistant.

All general SDK Assistant files are stored in `~/.da/`. This folder contains the `da` binary itself, a global SDK configuration file called `da.yaml`, and the downloaded SDK releases and related packages. On installation, the SDK Assistant will attempt to set up a symlink in `/usr/local/bin`.

#### 4.1.2 Understanding the SDK Assistant

The SDK Assistant is designed to make DAML development as easy and enjoyable as possible by helping with two key tasks:

**Initializing DAML SDK projects so you can use development tools** for project-specific application code. All SDK development should be confined within a DAML SDK project.

**Managing DAML SDK releases from Digital Asset** by periodically checking for updates. When an update is available, it is automatically downloaded and installed, keeping your SDK instance up to date. SDK releases contain development tools and libraries – the DAML compiler, the DAML Studio IDE, and a DAML ledger, for example.

---

**Note:** Because the SDK Assistant controls SDK releases and related tools, it can manage several versions of a tool and ensure that it uses the version compatible with the active project and other running tools. See [Managing SDK releases](#).

---

### 4.1.3 Using the SDK Assistant

The SDK Assistant is invoked with the `da` command in a terminal. If you do not add any arguments to the command, it will default to display a status message.

Use the `da --help` command to display a list of valid options and commands:

```
da --help

SDK Assistant - Version

Usage: da [-c|--config FILENAME] [-l|--log-level debug|info|warn|error]
          [--script] [--term-width ARG] [COMMAND]
SDK Assistant. Use --help for help.

Available options:
-h,--help           Show this help text
-c,--config FILENAME   Specify what config file to use.
-l,--log-level debug|info|warn|error
                      Set the log level. Default is 'warn'.
--script            Script mode -- skip auto-upgrades and similar.
--term-width ARG    Rendering width of the terminal.
-v,--version         Show version and exit

Available commands:
status              Show SDK environment overview
docs               Show SDK documentation
new                Create a new project from template
add                Add a template to the current project
project            Manage DAML SDK projects
template           Manage DAML SDK templates
upgrade            Upgrade to latest SDK version
list               List installed SDK versions
use                Set the default SDK version, downloading it if
                  necessary
uninstall          Remove DAML SDK versions or the complete DAML
SDK
  ↳ start            Start a given service
  ↳ restart          Restart a given service
  ↳ stop             Stop a given service
  ↳ feedback         Send us feedback!
  ↳ studio           Start DAML Studio in the current project
  ↳ navigator        Start Navigator (also runs Sandbox if needed)
  ↳ sandbox          Start Sandbox process in current project
  ↳ compile          Compile a DAML project into a DAR package
  ↳ path             Show the filesystem path of an SDK component
  ↳ run              Run the main executable of a package
  ↳ setup            Set up SDK environment (e.g. on install or
                    upgrade)
  ↳ subscribe        Subscribe for a namespace (of the template
                    repository).
```

(continues on next page)

(continued from previous page)

unsubscribe	Unsubscribe from a namespace (of the template repository).
config-help	Show config-file help
config	Query and manage configuration
changelog	Show the changelog of an SDK version
update-info	Show SDK Assistant update channel information

To get help for a particular command, use this command:

```
da <command> --help
```

**Example:**

```
da new --help

Usage: da new PROJECT_PATH
Create a new project

Available options:
-h,--help           Show this help text
PROJECT_PATH        Path to the new project. Name of the last folder
↳ will             be the name of the new project.
```

#### 4.1.4 Developing with the SDK Assistant

To begin, create a new DAML SDK project using the SDK Assistant. A project consists of a folder with a valid `da.yaml` file that, among other things, specifies which SDK release is being used. This is important because the release determines which versions of DAML and the DAML Sandbox the project code uses.

In the SDK Assistant, create a DA project with this command:

```
da new <project_path>
```

**Example:**

```
da new my-project
```

This example creates a project folder called `my-project` folder that is seeded with a basic folder structure.

Change to the new project directory.

**Example:**

```
cd my-project
```

In the project folder, use SDK Assistant functions. For example, if you have Visual Studio Code installed, open DAML Studio for your project using:

```
da studio
```

Use the following command to start the DAML Sandbox and the Navigator against the current project's DAML code:

```
da start
```

These services will run in the background. To see a list of running services, use this command:

```
da status
```

To stop any running services of the current project:

```
da stop
```

To restart:

```
da restart
```

The current SDK functionality is still basic. Future releases will improve and extend the current functionality and features to help you develop automation, UIs, and other types of functionality on top of your DAML application.

#### 4.1.5 Building DAML archives

The SDK Assistant can compile your DAML source code into a DAML archive (a `.dar` file). To do this, run:

```
da compile
```

This will compile the source code file set in your `da.yaml` configuration file (see [Configuring compilation](#)) and generate a `.dar` file in the directory `target` with the same name as your project.

#### 4.1.6 Managing SDK releases

The SDK Assistant will automatically check for updates and upgrade itself if there is a new version. This means that you will always have the latest version.

If you are working on several projects that have code that assumes different tool and component versions, you will need to have several SDK releases installed at the same time. Having different SDK releases available ensures all your projects will work without your having to upgrade all of them to use the latest libraries and tools. The SDK Assistant has commands for managing several SDK releases.

The **active** release is the one that is currently in use. For example, each DAML SDK project specifies the SDK release to use. When you are in a project, the specified release is the active one. The **default** release is the one that will be used when you start a new project or in other cases when an SDK release is needed and you are not in a project.

To list all installed SDK releases and see which are active and which is the default, use this command:

```
da list
```

To change the default release:

```
da use <release-version>
```

To upgrade to the latest SDK release manually:

```
da upgrade
```

To remove unused old releases:

```
da uninstall <release-version>
```

#### 4.1.7 Managing SDK templates

You can use the SDK Assistant to subscribe to templates, which are stored under a specific namespace. Often these are DAML templates, but not all of them are. There are two types of template:

Project templates, which are the starting point for a full project. These give you a folder with its own `da.yaml` file.

Add-on templates, which are the starting point for a sub-project. These give you a folder that lives inside a DAML SDK project folder. Add-on templates are not always DAML templates.

The Assistant comes with several DAML templates that are pre-made example modules, which show advanced uses of DAML and provide useful library functions.

To list available project templates, run:

```
da new
```

To list available add-on templates, run:

```
da add
```

---

**Note:** `da new` and `da add` may show undocumented or unsupported templates.

---

To subscribe to a namespace (so you can access the template inside it), run:

```
da subscribe <namespace>
```

To unsubscribe (remove from the list of namespaces which are checked if the listing commands are executed), run:

```
da unsubscribe <namespace>
```

Once you've subscribed to a project template's namespace, you can start a new project from the project template:

```
da new <namespace>/<project-template-name> <project-path>
```

Once you've subscribed to an add-on template's namespace, you can add the add-on template to your project:

```
da add <namespace>/<add-on-template-name> <target-folder>
```

For DAML templates, the new DAML files are added to your project inside the `daml` folder. To use them in your project, you need to add references to these files in the `daml/Main.daml` file with `import` statements. This is not done automatically.

#### 4.1.8 Running SDK tools

The SDK also comes with some packages that you can run and that can serve as important tools in the development of the project. You can run an executable package with the following command:

```
da run <package> [-- ARGS]
```

If you run the command above without the package argument, it will display a message saying that the package is missing and will show a list of all the available packages. The available packages currently are `damlc` and `navigator`.

The `damlc` package is a DAML compiler tool. The compiler functionality is currently intended only for internal use, so it is not described in this documentation.

The `navigator` package contains a browser-based tool that can also be used to interact with the ledger and is described in [Navigator](#).

#### 4.1.9 da.yaml configuration

The SDK Assistant uses a configuration file - `da.yaml` - that includes both global configuration and project configuration information. The SDK Assistant global `da.yaml` file is located in the SDK Assistant home folder (`~/.da/`). Project configuration in this file is ignored. Each project also has a `da.yaml` file, which specifies project configuration and can also override global configuration properties. To list the properties that can be specified in the configuration file, use this command:

```
da config-help
```

##### 4.1.9.1 Configuring parties

Some tools, like the Navigator, require parties to be configured before it is started. When using the SDK Assistant to start such services, configure parties in the `da.yaml` file for the project. The `project.parties` property takes a list of strings. For example:

```
...
project:
  sdk-version: '0.6.0'
  scenario: Main:example
  name: my-project
  source: daml/Main.daml
  parties:
    - OPERATOR
```

(continues on next page)

(continued from previous page)

- BANK1
- BANK2
- '007'

This can also be formatted in YAML as:

```
...
project:
  sdk-version: '0.6.0'
  scenario: Main:example
  name: my-project
  source: daml/Main.daml
  parties: ["OPERATOR", "BANK1", "BANK2", "007"]
```

#### 4.1.9.2 Configuring compilation

DAML compilation is configured with the following project variables:

**project.name** The name of the project.

**project.source** The path to the source code.

**project.output-path** The directory to store generated .dar files, the default is target.

The generated .dar file will be stored in \${project.output-path}/\${project.name}.dar.

#### 4.1.10 Uninstalling DAML SDK

If for any reason the DAML SDK needs to be removed use:

```
da uninstall all
```

This command will ask for confirmation and remove the complete DAML SDK home folder (~/.da/) and the da symlink in /usr/local/bin created upon installation.

## 4.2 DAML Sandbox

The DAML Sandbox, or Sandbox for short, is an in-memory ledger that enables rapid application prototyping by simulating the Digital Asset Distributed Ledger. It is a stand-alone JVM application utilizing the same components as the runtime platform.

You can start DAML Sandbox together with [Navigator](#) using the `da start` command in a DAML SDK project. This command will compile the DAML file and its dependencies as specified in the `da.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 `da.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 `da sandbox`.

Sandbox can also be run manually as in this example:

```
$ da run sandbox -- Main.dar --scenario Main:example
/ \ / \
\ \ / \ ` / \ / \ / \ / \
/ \ / \ , / \ / \ , / \ . / \ / \ 
initialized sandbox with ledger-id = sandbox-16ae201c-b2fd-45e0-af04-
˓→c61abe13fed7, port = 8080, dar file = DAR files at List(/Users/
˓→donkeykong/temp/da-sdk/test/Main.dar), time mode = Static, daml-engine =
˓→{}  
Initialized Static time provider, starting from 1970-01-01T00:00:00Z  
listening on localhost:8080
```

Here, `da run 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 `:`. The scenario is used for testing and development; it is not run in production.

### 4.2.1 Command-line reference

Sandbox requires the names of the input .dar or .dalf files as arguments to start. The available command line options are listed here:

```
-p, --port <value>           Sandbox service port. Defaults to 8080.  
-a, --address <value>       Sandbox service host. Defaults to binding on all  
→addresses.  
--dalf                      This argument is present for backwards  
→compatibility. DALF and DAR archives are now identified by their  
→extensions.  
--static-time                Use static time, configured with TimeService  
→through gRPC.  
-w, --wall-clock-time       Use wall clock time (UTC). When not provided,  
→static time is used.  
--no-parity                 Disables Ledger Server parity mode. Features which  
→are not supported by the Platform become available.  
--scenario <value>          If set, the sandbox will execute the given  
→scenario on startup and store all the contracts created by it. Two  
→formats are supported: Module.Name:Entity.Name (preferred) and Module.  
→Name.Entity.Name (deprecated, will print a warning when used).  
--daml-lf-archive-recursion-limit <value>  
                           Set the recursion limit when decoding DAML-LF  
→archives (.dalf files). Default is 1000  
<archive>...                Daml archives to load. Either in .dar or .dalf  
→format. Only DAML-LF v1 Archives are currently supported.  
--help                       Print the usage text
```

## 4.2.2 Performance benchmarks

Sandbox's performance depends on many different factors. These include the hardware it is run on, the size of the specific models used in the test, and the characteristics of the applications used on the client side of the Ledger API. This means you should treat the numbers below merely as an indication of Sandbox's capabilities.

In the tests performed to date, Sandbox has been shown to:

- start in less than 3s
- create 100,000 contracts within 12s window
- consume less than 200,000MB of memory for the benchmark 100,000 contracts
- support simultaneous connections from 10 clients all sending commands at around 1'000 contracts per second.

## 4.3 Navigator

The Navigator is a front-end that you can use to connect to any Digital Asset ledger and inspect and modify the ledger. You can use it during DAML development to explore the flow and implications of the DAML models.

The first sections of this guide cover use of the Navigator with the DAML SDK. Refer to [Advanced usage](#) for information on using Navigator outside the context of the SDK.

### 4.3.1 Navigator functionality

Connect Navigator to any Digital Asset ledger and use it to:

- View templates
- View active and archived contracts
- Exercise choices on contracts
- Advance time (This option applies only when using Navigator with the DAML Sandbox ledger.)

### 4.3.2 Installing and starting Navigator

Navigator ships with the DAML SDK. To launch it:

1. Start Navigator via a terminal window running [SDK Assistant](#) by typing `da 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 `da start` you will see the Navigator URL. By default it will be <http://localhost:7500/>. However, if port 7500 is taken, a different port will be picked.

---

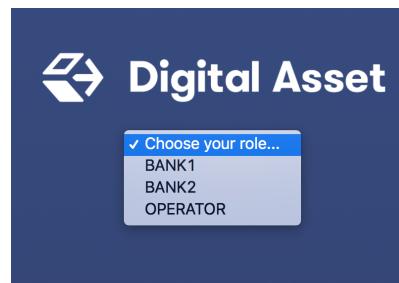
**Note:** Navigator is compatible with these browsers: Safari, Chrome, or Firefox.

---

For information on how to launch and use Navigator outside of the SDK, see [Advanced usage](#) below.

### 4.3.3 Choosing a party / changing the party

The ledger is a record of transactions between authorized participants on the distributed network. Before you can interact with the ledger, you must assume the role of a particular party. This determines the contracts that you can access and the actions you are permitted to perform on the ledger. The first step in using Navigator is to use the drop-down list on the Navigator home screen to select from the available parties.



**Note:** The party choices are configured on startup. (Refer to [SDK Assistant](#) 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_eedd77f22bcba4cf687bd282040f...	2017-03-06T14:00:00Z	(blue circle)
9:2_	fxSwap_t4@FX_Market_3_e48300987eff7945102c59e14ef72696350e84bcd...	2017-03-06T14:00:00Z	(blue circle)
6:1_	cashlou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d...	2017-03-06T14:00:00Z	(blue circle)
9:8_	lockedCashlou_t2@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d...	2017-03-06T14:00:00Z	(blue circle)
10:11_	cashlou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d...	2017-03-06T14:00:00Z	(blue circle)
10:5_	cashlou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d...	2017-03-06T14:00:00Z	(blue circle)
8:1_	fxSwapOffer_t6@FX_Market_4_338d5ea7db6c83764ebd9a8835e9d54521b7...	2017-03-06T14:00:00Z	(blue circle)

To change the active party:

1. Click the name of the current party in the top right corner of the screen.
2. On the home screen, select a different party.



You can act as different parties in different browser windows. Use Chrome's profile feature <https://support.google.com/chrome/answer/2364824> and sign in as a different party for each Chrome profile.

#### 4.3.4 Logging out

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

#### 4.3.5 Viewing templates or contracts

DAML contract templates are models that contain the agreement statement, all the applicable parameters, and the choices that can be made in acting on that data. They specify acceptable input and the resulting output. A contract template contains placeholders rather than actual names, amounts, dates, and so on. In a contract *instance*, the placeholders have been replaced with actual data.

The Navigator allows you to list templates or contracts, view contracts based on a template, and view template and contract details.

##### 4.3.5.1 Listing templates

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

TEMPLATE ID	# CONTRACTS
fxLegBuyerCommitted_t5@FX_Market_2_eedd7f22bcba4fcf687bd282040fcfde2596e4997c079d303d0026d03b...	1
fxSwapAllegeRight_t7@FX_Market_5_c5f76e42a5d7e2c9af73f2ab88599e1f06fd7218fc32c3e0d3f4473c62ef9e0f...	1
cashSettlement_t0@FX_Market_0_a2c769c7083fdc3598ae60f9f6b1f7e709d7169d26ec2d8aba20543195ff3aa...	1
fxSwapOffer_t6@FX_Market_4_338d5ea7db6c83764ebd9a8835e9d54521b7126402bf15f10fe0258e9765f2e...	1
cashlou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d3913a8b00b791faf55e669...	1
lockedCashlou_t2@FX_Market_1_a4327da785f16317aa1f5b78576aa09a7e914545d3913a8b00b791faf55e669...	1
fxSwap_t4@FX_Market_3_e48300987aff7945102c59e14e72686350e84dbc1453ab30accae20dc09ec...	1
cashlouissuanceRight_t3@FX_Market_6_83728498ad29b7ad030a89f11e966b6abc732010712a2032a35793c7104...	1
emptyContract_t0@Utilities_0_0e62858e278cd9e38dba8682e3b83bb62613976371975c81ca5fb24e62e28ad1...	1

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

#### 4.3.5.2 Listing contracts

To view a list of available contracts, choose **Contracts** in the left pane.

ID	TEMPLATE ID	TIME	CHOICE
9:9	fxLegBuyerCommitted_t5@FX_Market_2_eedd77f22bcba4cf687bd282040f...	2017-03-06T14:00:00Z	
9:2	fxSwap_t4@FX_Market_3_e48300987aff7945102c59e14ef72696350e84bd...	2017-03-06T14:00:00Z	
6:1	cashlou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d...	2017-03-06T14:00:00Z	
9:8	lockedCashlou_t2@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d...	2017-03-06T14:00:00Z	
10:11	cashlou_t1@FX_Market_1_a4327ada785f16317aa1f5b78576aa09a7e914545d...	2017-03-06T14:00:00Z	
10:5	cashlou_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.

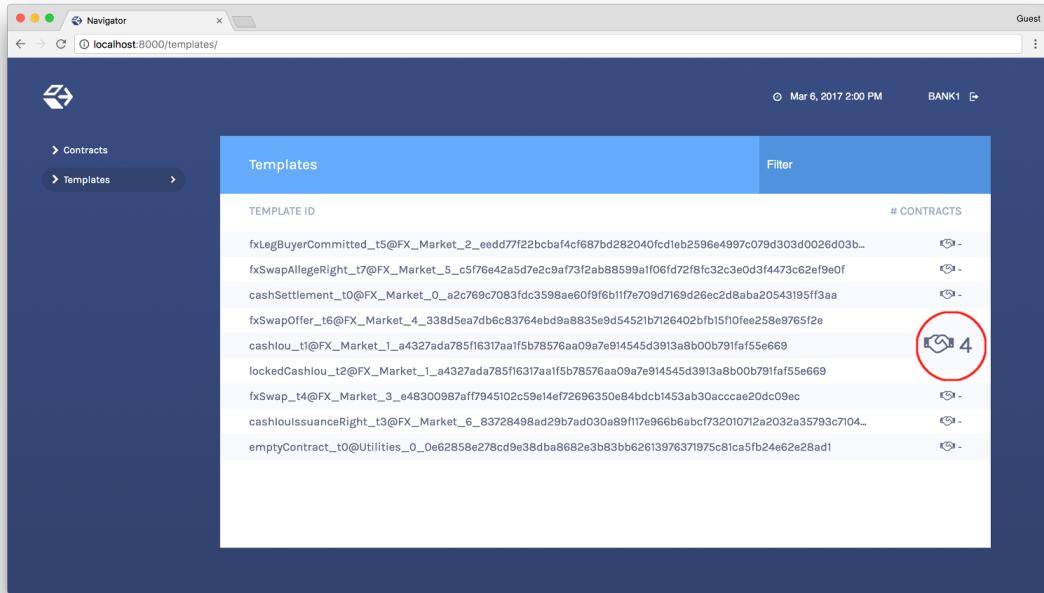
Filter the displayed contracts by entering text in the **Filter** field at the top right. Use the **Include Archived** checkbox at the top to include or exclude archived contracts.

#### 4.3.5.3 Viewing contracts based on a template

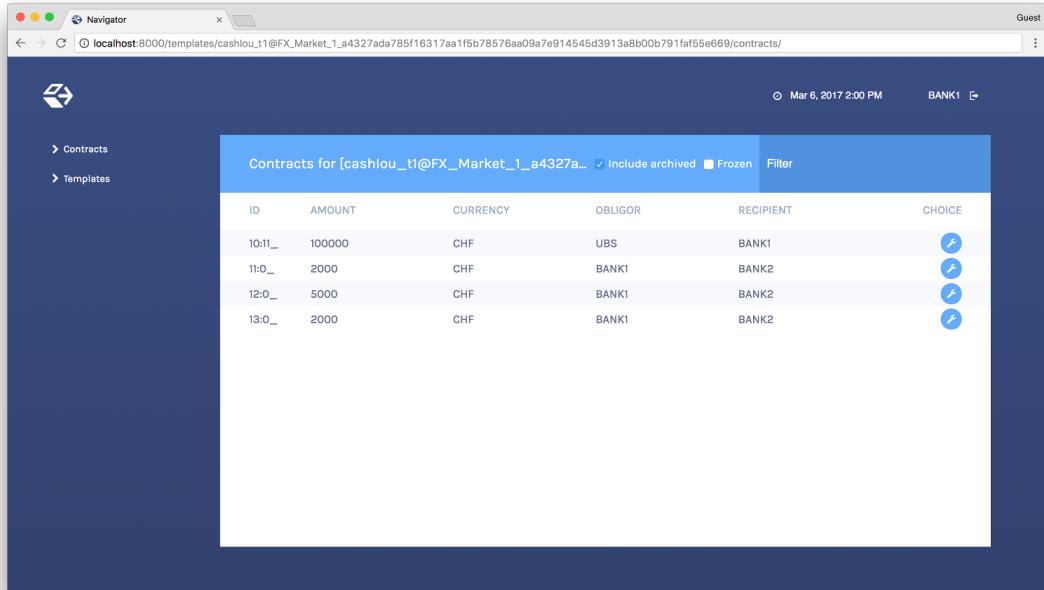
You can also view the list of contracts that are based on a particular template.

1. You will see icons to the right of template IDs in the template list with a number indicating how many contracts are based on this template.
2. Click the number to display a list of contracts based on that template.

#### Number of Contracts



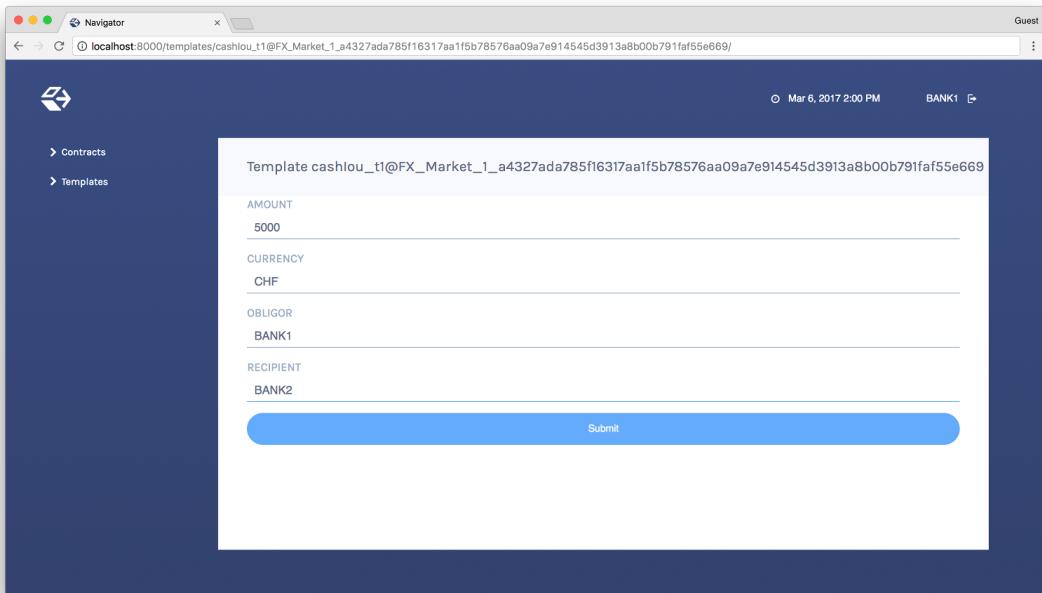
## List of Contracts



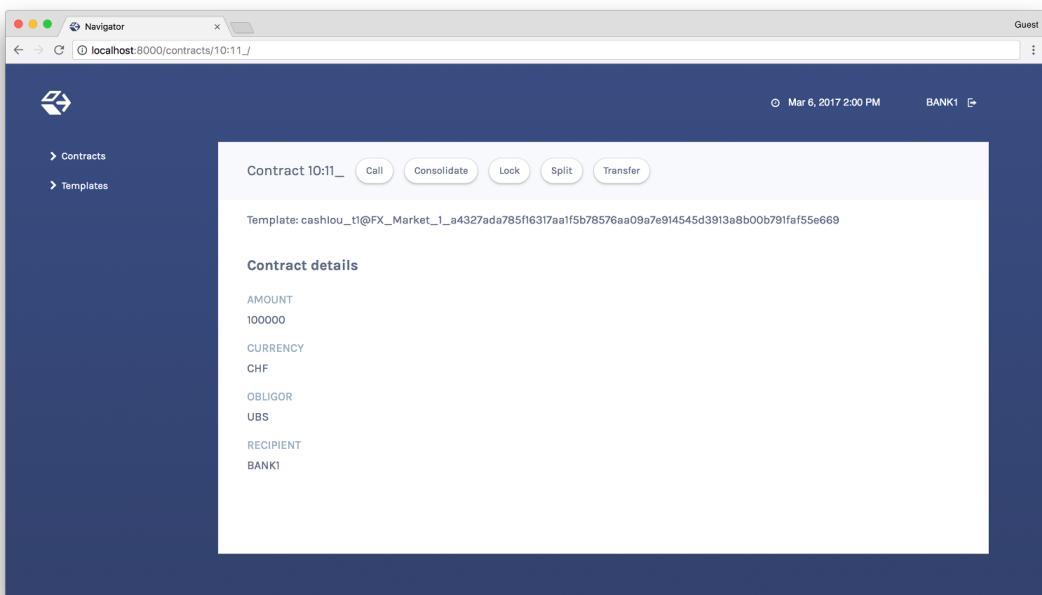
### 4.3.5.4 Viewing template and contract details

To view template or contract details, click on a template or contract in the list. The template or contracts detail page is displayed.

#### Template Details



## Contract Details



### 4.3.6 Using Navigator

#### 4.3.6.1 Creating contracts

Contracts in a ledger are created automatically when you exercise choices. In some cases, you create a contract directly from a template. This feature can be particularly useful for testing and experimenting during development.

To create a contract based on a template:

1. Navigate to the template detail page as described above.

2. Complete the values in the form
3. Choose the **Submit** button.

The screenshot shows a web application window titled "Navigator". The address bar indicates the URL is "localhost:8000/templates/cashlou\_t1@FX\_Market\_1\_a4327ada785f16317aa1f5b78576aa09a7e914545d3913a8b00b791faf55e669". The top right corner shows "Guest". The main content area displays a form titled "Template cashlou\_t1@FX\_Market\_1\_a4327ada785f16317aa1f5b78576aa09a7e914545d3913a8b00b791faf55e669". The form contains four fields: "AMOUNT" with value "0", "CURRENCY" with value "Text", "OBLIGOR" with value "Party", and "RECIPIENT" with value "Party". A blue "Submit" button is located at the bottom of the form.

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

While loading



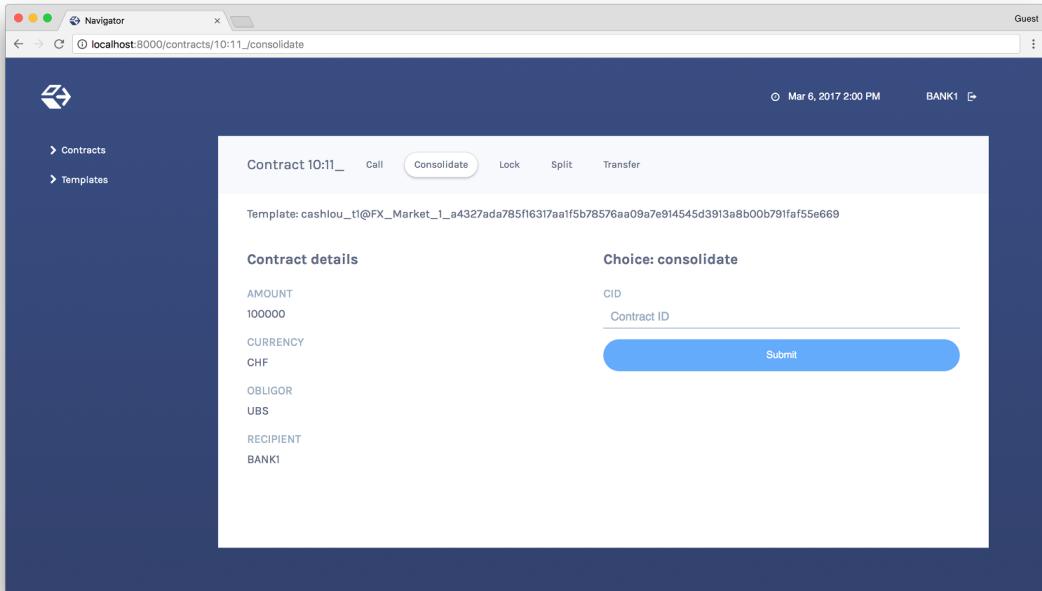
When committed to the ledger



#### 4.3.6.2 Exercising choices

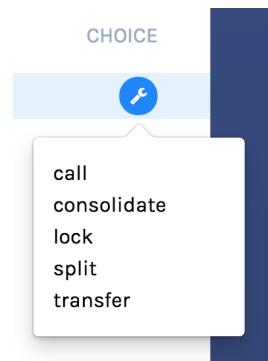
To exercise a choice:

1. Navigate to the contract details page (see above).
2. Click the choice you want to exercise in the choice list.
3. Complete the form.
4. Choose the **Submit** button.



Or

1. Navigate to the choice form by clicking the wrench icon in a contract list.
2. Select a choice.



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

#### 4.3.6.3 Advancing time

It is possible to advance time against the DAML Sandbox. (This is not true of the Digital Asset ledger.) This advance-time functionality can be useful when testing, for example, when entering a trade on one date and settling it on a later date.

To advance time:

1. Click on the ledger time indicator in the navbar at the top of the screen.
2. Select a new date / time.
3. Choose the **Set** button.



## 4.3.7 Advanced usage

### 4.3.7.1 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:

```
export const version = {
  schema: 'navigator-config',
  major: 1,
  minor: 0,
};

export const customViews = (userId, party, role) => ({
  customview1: {
    type: "table-view",
    title: "Filtered contracts",
    source: {
      type: "contracts",
      filter: [
        {
          field: "id",
          value: "1",
        }
      ],
      search: "",
      sort: [
        {
          field: "id",
          direction: "ASCENDING"
        }
      ]
    },
  }
},
```

(continues on next page)

(continued from previous page)

```
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>).

#### 4.3.7.2 Using Navigator outside the SDK

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

---

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

---

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

```
da path navigator
```

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

```
da run navigator
```

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

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

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

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

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" }
}
```

---

**Note:** The password is used only if you activate the `--require-password` flag. This feature is only intended for demonstration purposes and may be removed in the future.

---

#### 4.3.7.3 Using Navigator with the Digital Asset ledger

By default, Navigator is configured to use an unencrypted connection to the ledger. To run Navigator against a secured Digital Asset Ledger, configure TLS certificates using the `--pem`, `--crt`, and `--cacrt` command line parameters. Details of these parameters are explained in the command line help:

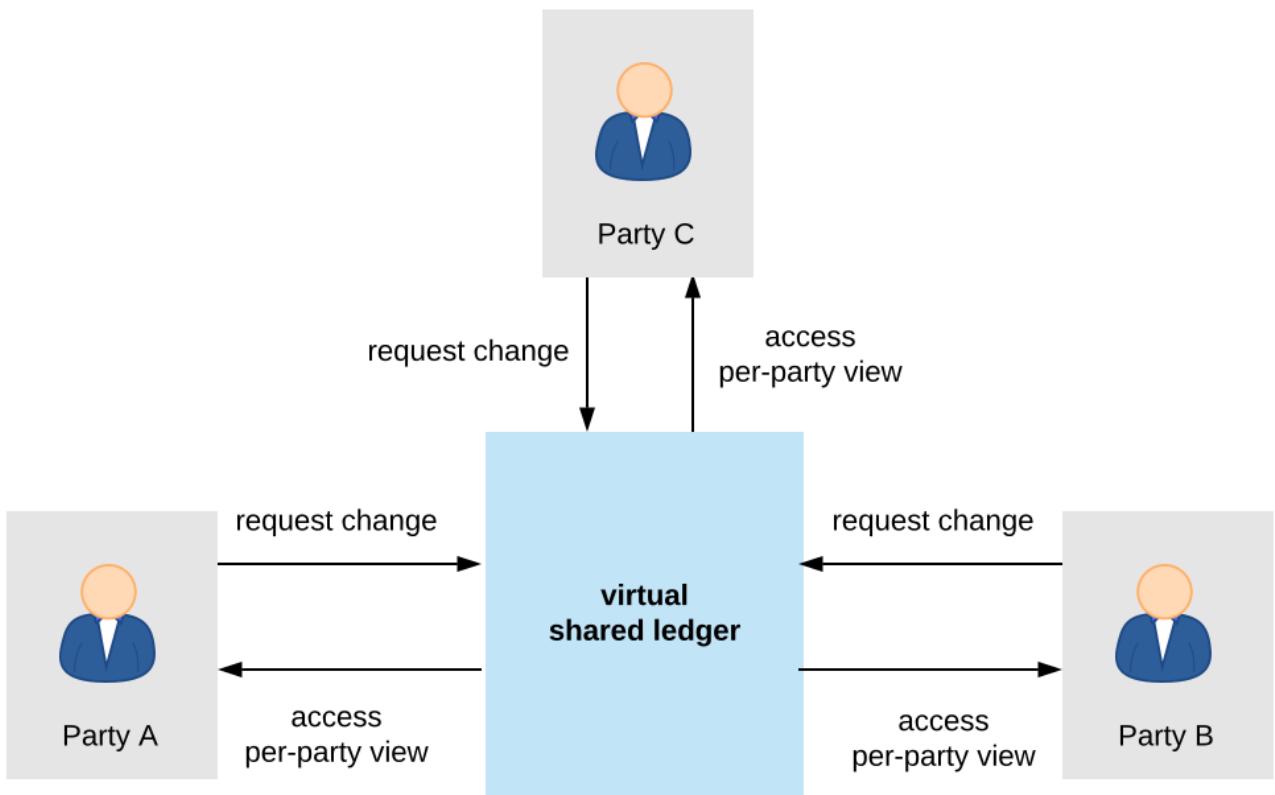
```
da run navigator -- --help
```

## Chapter 5

# Background Concepts

### 5.1 DA Ledger Model

The Digital Asset Platform enables multi-party workflows by providing parties with a virtual shared ledger, which encodes the current state of their shared contracts, written in DAML. At a high level, the interactions are visualized as follows:



The DA ledger model defines:

1. what the ledger looks like - the structure of DA ledgers
2. who can request which changes - the integrity model for DA ledgers
3. who sees which changes and data - the privacy model for DA ledgers

The below sections review these concepts of the ledger model in turn. They also briefly describe the

link between DAML and the model.

## 5.1.1 Structure

This section looks at the structure of a DA ledger and the associated ledger changes. The basic building blocks of changes are actions, which get grouped into transactions.

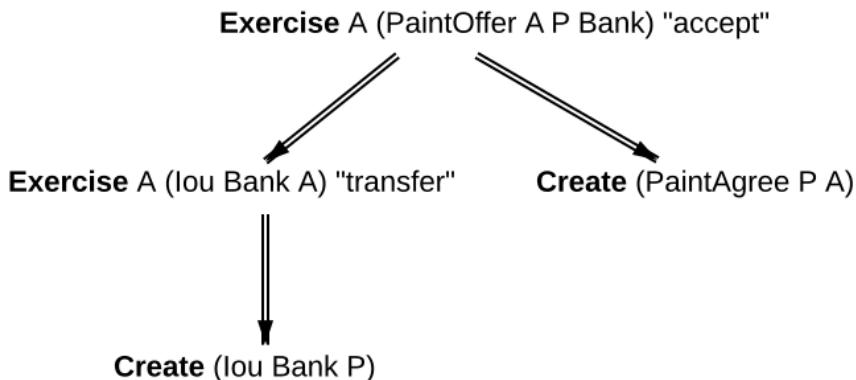
### 5.1.1.1 Actions and Transactions

One of the main features of the DA ledger model is a *hierarchical action structure*.

This structure is illustrated below on a toy example of a multi-party interaction. Alice (A) gets some digital cash, in the form of an I-Owe-You (IOU for short) from a bank, and she needs her house painted. She gets an offer from a painter (P) to paint her house in exchange for this IOU. Lastly, A accepts the offer, transferring the money and signing a contract with P, whereby he is promising to paint her house.

This acceptance can be viewed as A exercising her right to accept the offer. Her acceptance has two consequences. First, A transfers her IOU, that is, exercises her right to transfer the IOU, after which a new IOU for P is created. Second, a new contract is created that requires P to paint A's house.

Thus, the acceptance in this example is reduced to two types of actions: (1) creating contracts, and (2) exercising rights on them. These are also the two main kinds of actions in the DA ledger model. The visual notation below records the relations between the actions during the above acceptance.



Formally, an **action** is one of the following:

1. a **Create** action on a contract, which records the creation of the contract
2. an **Exercise** action on a contract, which records that one or more parties have exercised a right they have on the contract, and which also contains:

1. An associated set of parties called **actors**. These are the parties who perform the action.
2. An exercise **kind**, which is either **consuming** or **non-consuming**. Once consumed, a contract cannot be used again (for example, the painter should not be able to accept the offer twice). Contracts exercised in a non-consuming fashion can be reused.
3. A list of **consequences**, which are themselves actions. Note that the consequences, as well as the kind and the actors, are considered a part of the exercise action itself. This

nesting of actions within other actions through consequences of exercises gives rise to the hierarchical structure. The exercise action is the **parent action** of its consequences.

3. a **Fetch** action on a contract, which demonstrates that the contract exists and is in force at the time of fetching. The action also contains **actors**, the parties who fetch the contract. A **Fetch** behaves like a non-consuming exercise with no consequences, and can be repeated.

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,  $s \mid t$  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.

```

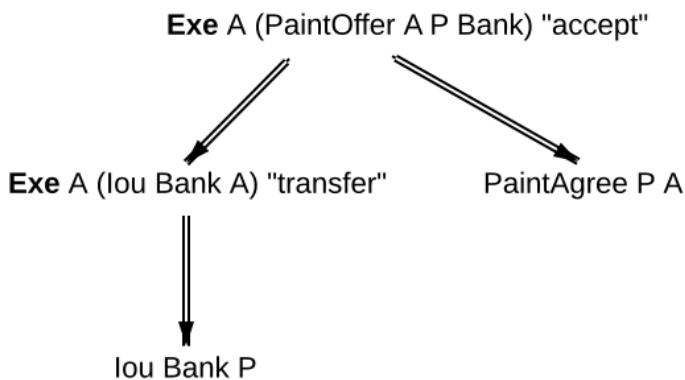
Action      ::= 'Create' contract
              | 'Exercise' party* contract Kind Transaction
              | 'Fetch' party* contract
Transaction ::= Action*
Kind        ::= 'Consuming' | 'NonConsuming'

```

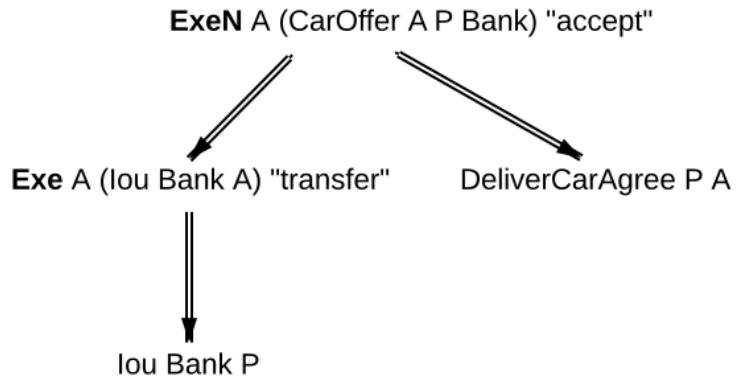
The visual notation presented earlier captures actions precisely with conventions that:

1. **Exercise** denotes consuming, **ExerciseN** non-consuming exercises, and **Fetch** a fetch.
2. double arrows connect exercises to their consequences, if any.
3. the consequences are ordered left-to-right.
4. to aid intuitions, exercise actions are annotated with suggestive names like accept or transfer. Intuitively, these correspond to names of DAML choices, but they have no semantic meaning.

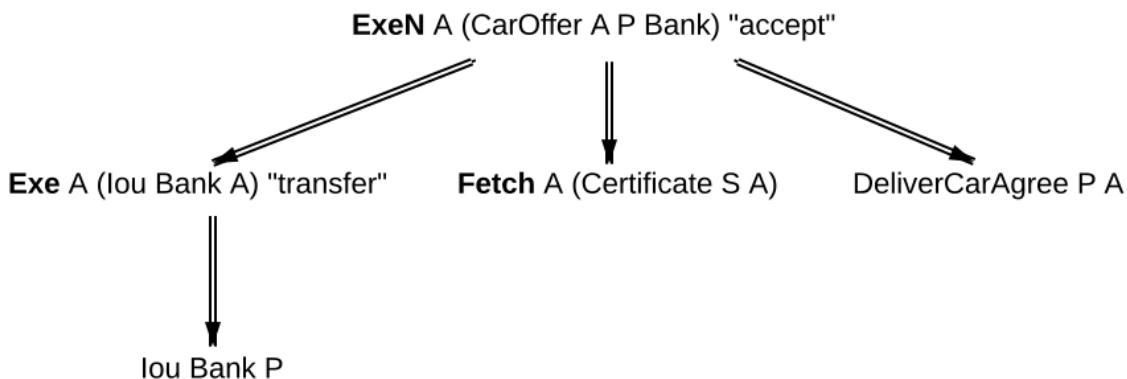
An alternative shorthand notation, shown below uses the abbreviations **Exe** and **ExeN** for exercises, and omits the **Create** labels on create actions.



To show an example of a non-consuming exercise, consider a different offer example with an easily replenishable subject. For example, if  $P$  was a car manufacturer, and  $A$  a car dealer,  $P$  could make an offer that could be accepted multiple times.



To see an example of a fetch, we can extend this example to the case where P produces exclusive cars and allows only certified dealers to sell them. Thus, when accepting the offer, A has to additionally show a valid quality certificate issued by some standards body S.



In the paint offer example, the underlying type of contracts consists of three sorts of contracts:

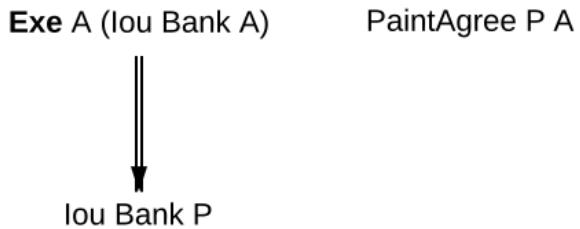
**PaintOffer houseOwner painter obligor** Intuitively an offer by which the painter proposes to the house owner to paint her house, in exchange for a single IOU token issued by the specified obligor.

**PaintAgree painter houseOwner** Intuitively a contract whereby the painter agrees to paint the owner's house

**IoU obligor owner** An IOU token from an obligor to an owner (for simplicity, the token is of unit amount).

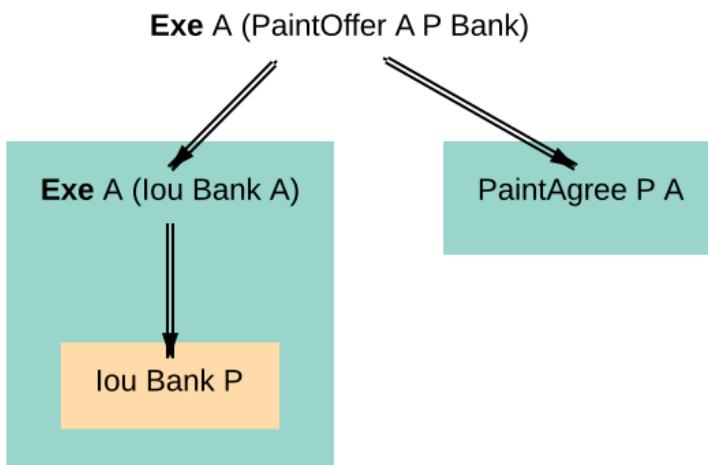
In practice, multiple IOU contracts would exist between the same obligor and owner, in which case each contract should have a unique identifier. However, in this section, each contract only appears once, allowing us to drop the notion of identifiers for simplicity reasons.

A **transaction** is a list of actions. Thus, the consequences of an exercise form a transaction. In the example, the consequences of the Alice's exercise form the following transaction, where actions are again ordered left-to-right.

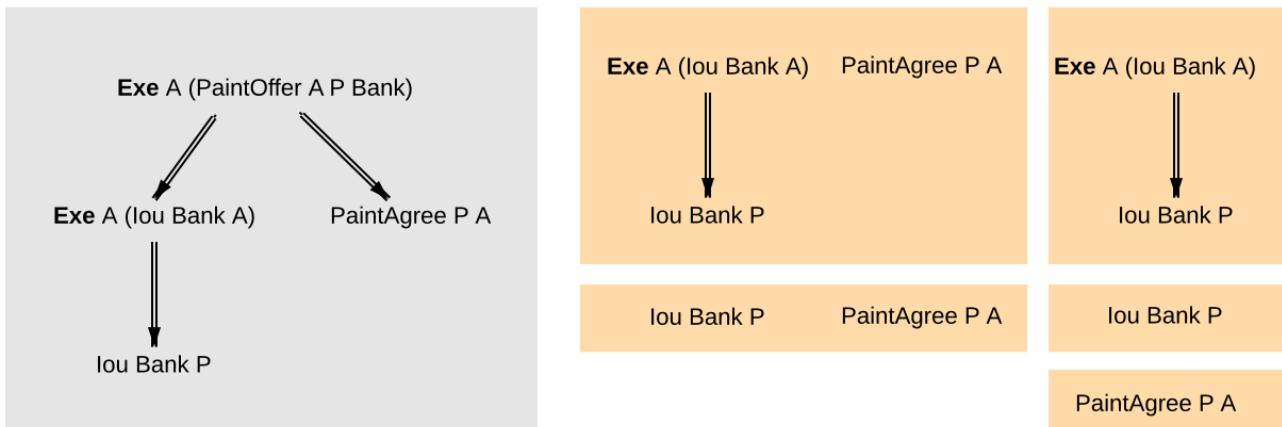


For an action act, its **proper subactions** are all actions in the consequences of act, together with all of their proper subactions. Additionally, act is a (non-proper) **subaction** of itself.

The subaction relation is visualized below. Both the green and yellow boxes are proper subactions of Alice's exercise on the paint offer. Additionally, the creation of *Iou Bank P* (yellow box) is also a proper subaction of the exercise on the *Iou Bank A*.



Similarly, a **subtransaction** of a transaction is either the transaction itself, or a **proper subtransaction**: a transaction obtained by removing at least one action, or replacing it by a subtransaction of its consequences. For example, given the the transaction consisting of just one action, the paint offer acceptance, the image below shows all its proper subtransactions on the right (yellow boxes).



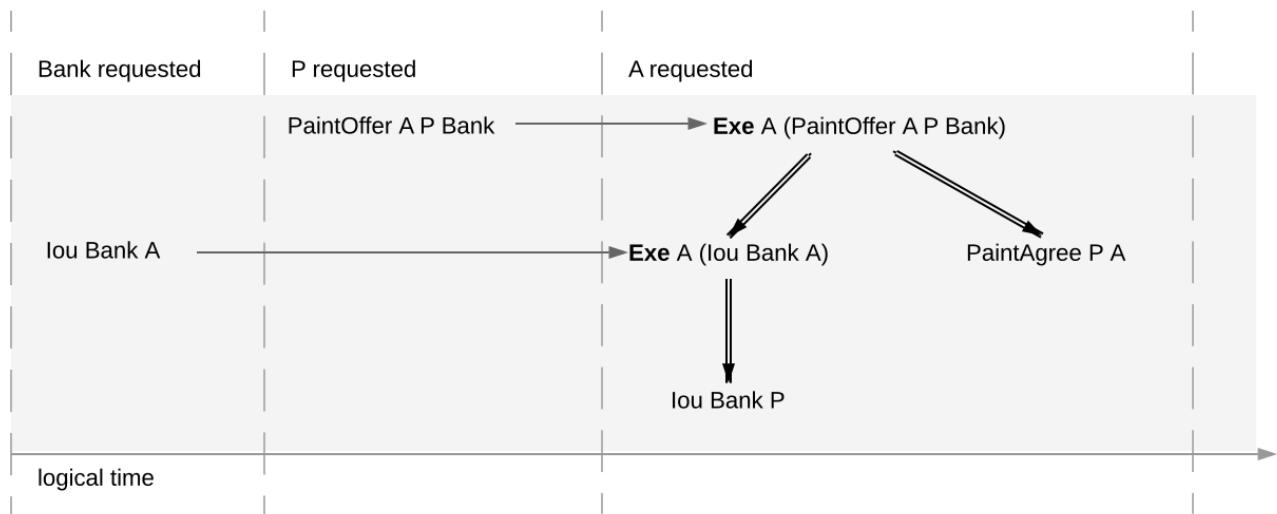
### 5.1.1.2 Ledgers

The transaction structure records the contents of the changes, but not who requested them. This information is added by the notion of a **commit**: a transaction paired with the parties that requested it, called the **requesters** of the commit. In the ledger model, a commit is allowed to have multiple requesters, although the current DA Platform API offers the request functionality only to individual parties. Given a commit  $(p, tx)$  with transaction  $tx = act_1, \dots, act_n$ , every  $act_i$  is called a **top-level action** of the commit. A **ledger** is a sequence of commits. A top-level action of any ledger commit is also a top-level action of the ledger.

The following EBNF grammar summarizes the structure of commits and ledgers:

```
Commit ::= party Transaction
Ledger ::= Commit*
```

A DA ledger thus represents the full history of all actions taken by parties.<sup>1</sup> Since the ledger is a sequence (of dependent actions), it induces an order on the commits in the ledger. Visually, a ledger can be represented as a sequence growing from left to right as time progresses. Below, dashed vertical lines mark the boundaries of commits, and each commit is annotated with its requester(s). Arrows link the create and exercise actions on the same contracts. These additional arrows highlight that the ledger forms a **transaction graph**. For example, the aforementioned house painting scenario is visually represented as follows.



The definitions presented here are all the ingredients required to record the interaction between parties in a DA ledger. That is, they address the first question: what do changes and ledgers look like?. To answer the next question, who can request which changes, a precise definition is needed of which ledgers are permissible, and which are not. For example, the above paint offer ledger is intuitively permissible, while all of the following ledgers are not.

The next section discusses the criteria that rule out the above examples as invalid ledgers.

### 5.1.2 Integrity

This section addresses the question of who can request which changes.

---

Calling such a complete record ledger is standard in the distributed ledger technology community. In accounting terminology, this record is closer to a journal than to a ledger.

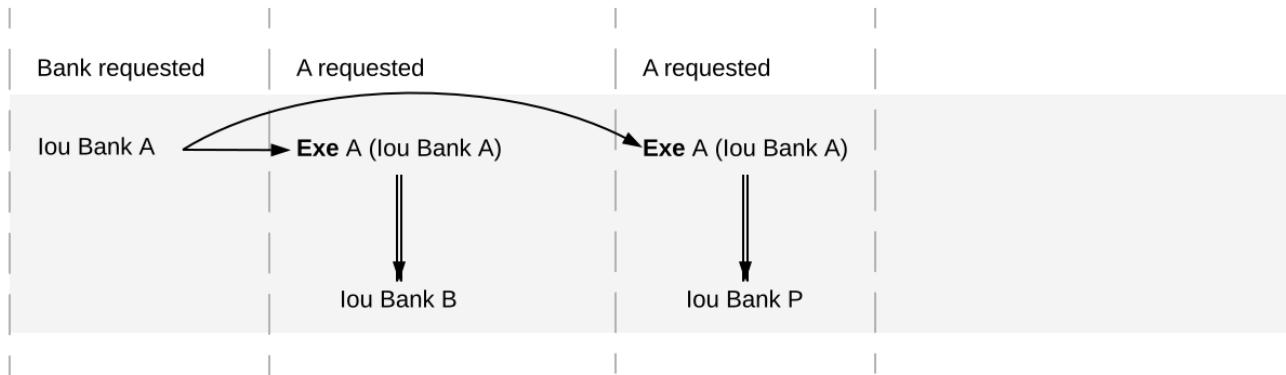


Fig. 1: Alice spending her IOU twice (double spend), once transferring it to B and once to P.

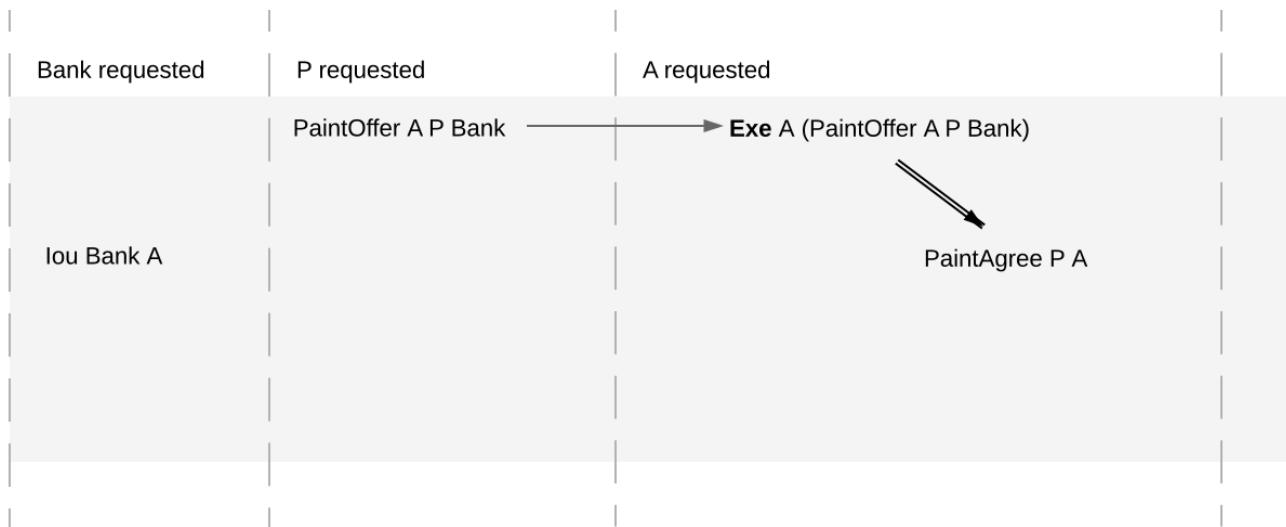


Fig. 2: Alice changing the offer's outcome by removing the transfer of the *iou*.



Fig. 3: An obligation imposed on the painter without his consent.

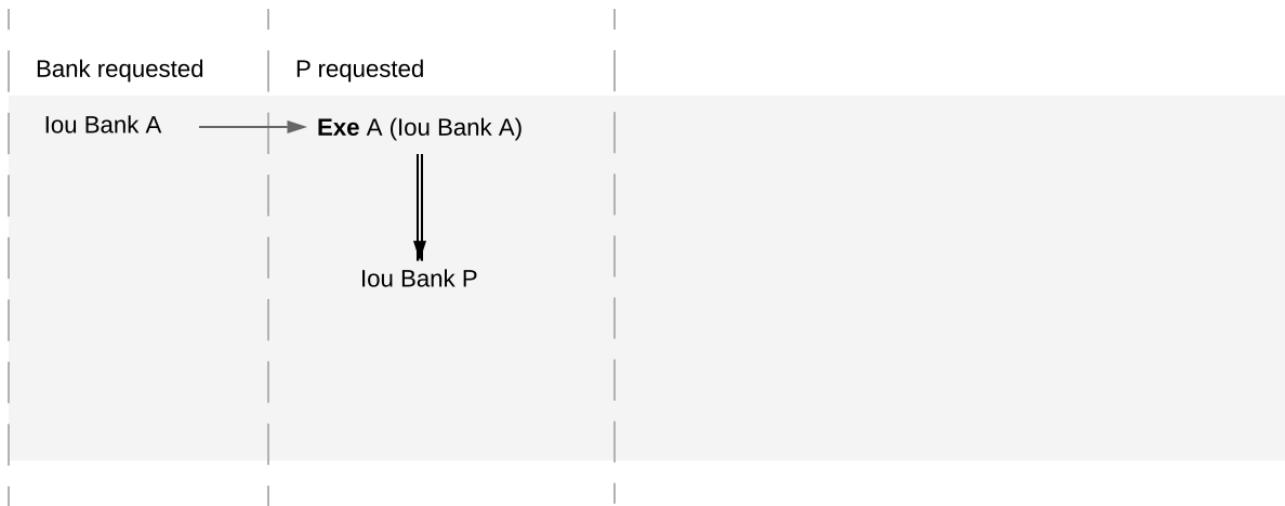


Fig. 4: Painter stealing Alice's IOU. Note that the ledger would be intuitively permissible if it was Alice performing the last commit.

### 5.1.2.1 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 on inactive contracts are not allowed, i.e. contracts that have not yet been created or have already been consumed by an exercise.

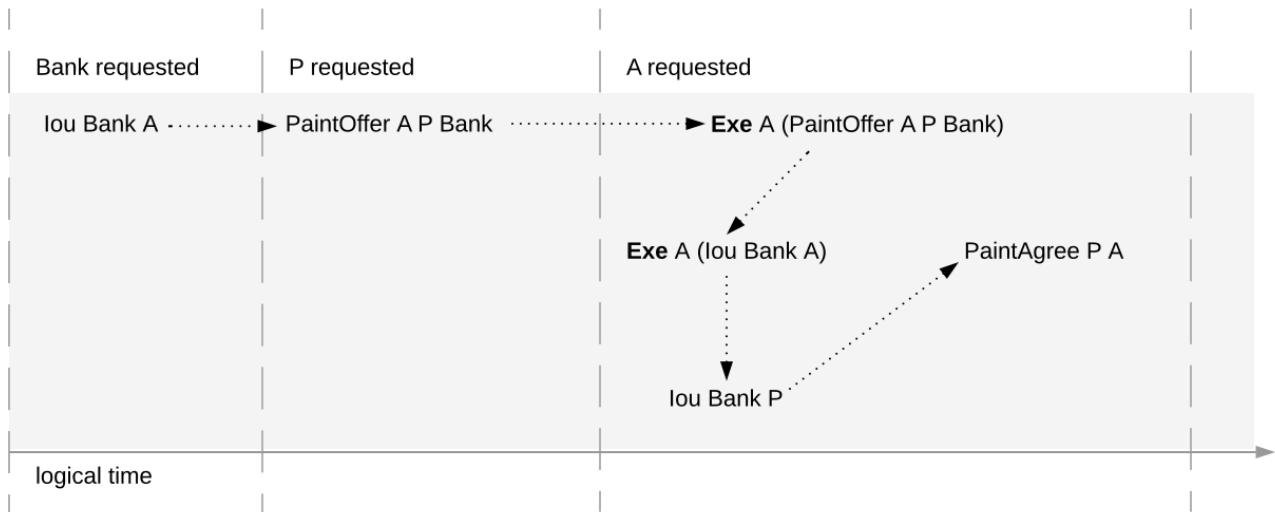
**Conformance** Only a restricted set of actions is allowed on a given contract.

**Authorization** The parties who may request a particular change are restricted.

Only the last of these conditions is dependent on the party (or parties) requesting the change; the other two are general.

### 5.1.2.2 Consistency

Intuitively, consistency requires contracts to be created before they are used, and that they cannot be used once they are consumed. 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:

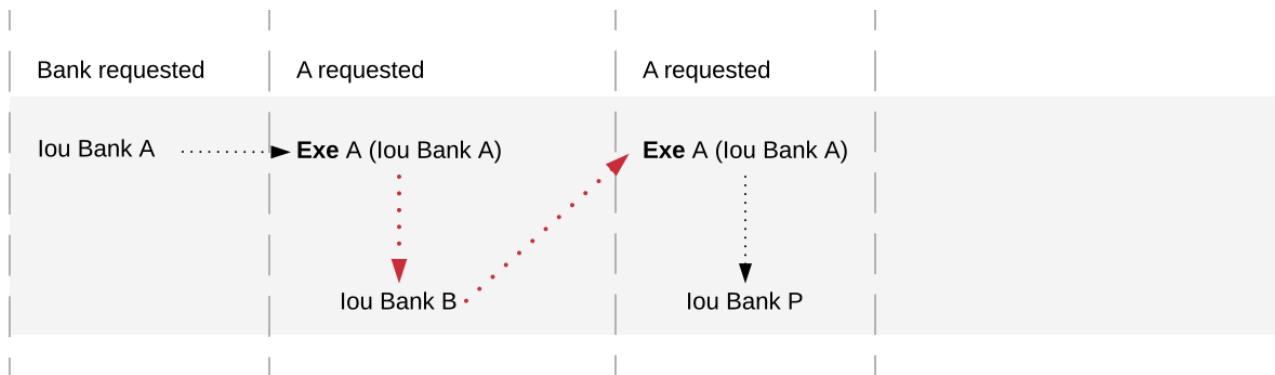


In the image, an action *act* happens before action *act'* if there is a (non-empty) path from *act* to *act'*. Then, *act'* happens after *act*. A ledger is **consistent for a contract *c*** if all of the following holds for all actions *act* on *c*:

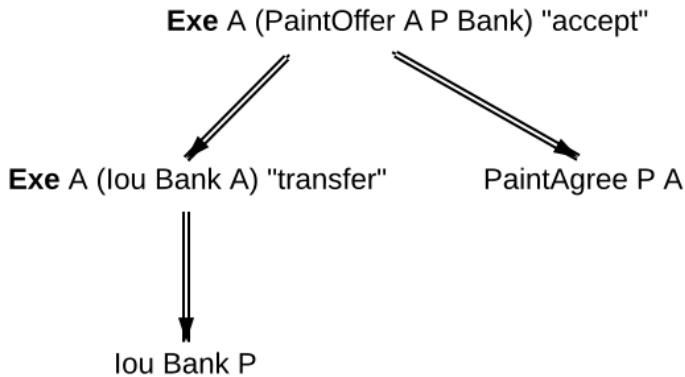
1. either *act* is itself **Create *c*** or a **Create *c*** happens before *act*
2. *act* does not happen before any **Create *c*** action
3. *act* does not happen after any exercise consuming *c*.

A ledger is **consistent** if it is consistent for all contracts.

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 consistency criteria.



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, because *PaintOffer A P Bank* and the *iou Bank A* contracts are used without being created before:



However, the transaction can still be appended to a ledger that creates these contracts and yield a consistent ledger. Such transactions are said to be internally consistent, and contracts such as the *PaintOffer A P Bank* 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** 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 c.

**Definition input contract** For an internally consistent transaction, a contract c is an **input contract** of the transaction if the transaction contains an **Exercise** or a **Fetch** action on c but not a **Create c** action.

**Definition output contract** For an internally consistent transaction, a contract c is an **output contract** of the transaction if the transaction contains a **Create c** action, but no a consuming **Exercise** action on c.

Note that the input and output contracts are undefined for transactions that are not internally consistent. The image below shows some examples of internally consistent and inconsistent transactions.

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 consuming. 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 was;
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 nonexistent. The figures below visualize the state of different contracts at all points in the example ledger.

The notion of order can be defined on all the different ledger structures: actions, transactions, lists of transactions, and ledgers. Thus, the notions of (internal) consistency, inputs and outputs, and contract state can also all be defined on all these structures. The **active contract set** of a ledger is the set of all contracts that are active on the ledger. For the example above, it consists of contracts *iou Bank P* and *PaintAgree P A*.

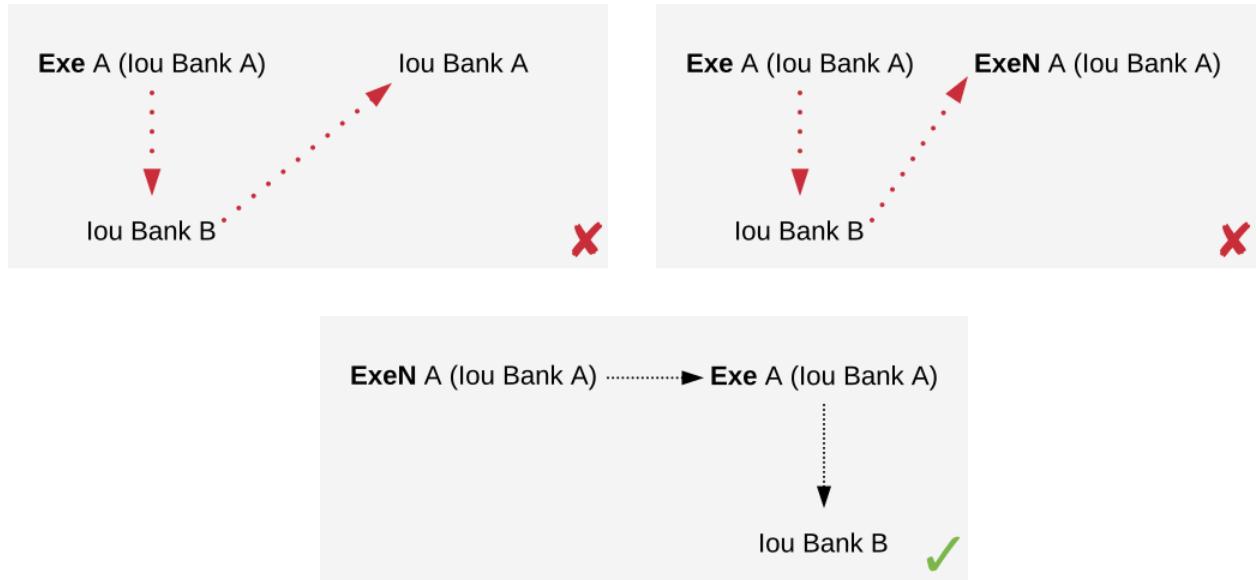


Fig. 5: The first two transactions violate the conditions of internally consistency. The first transaction creates the *Iou* after exercising it consumingly, violating both conditions. The second transaction contains a (non-consuming) exercise on the *Iou* after a consuming one, violating the second condition. The last transaction is internally consistent.

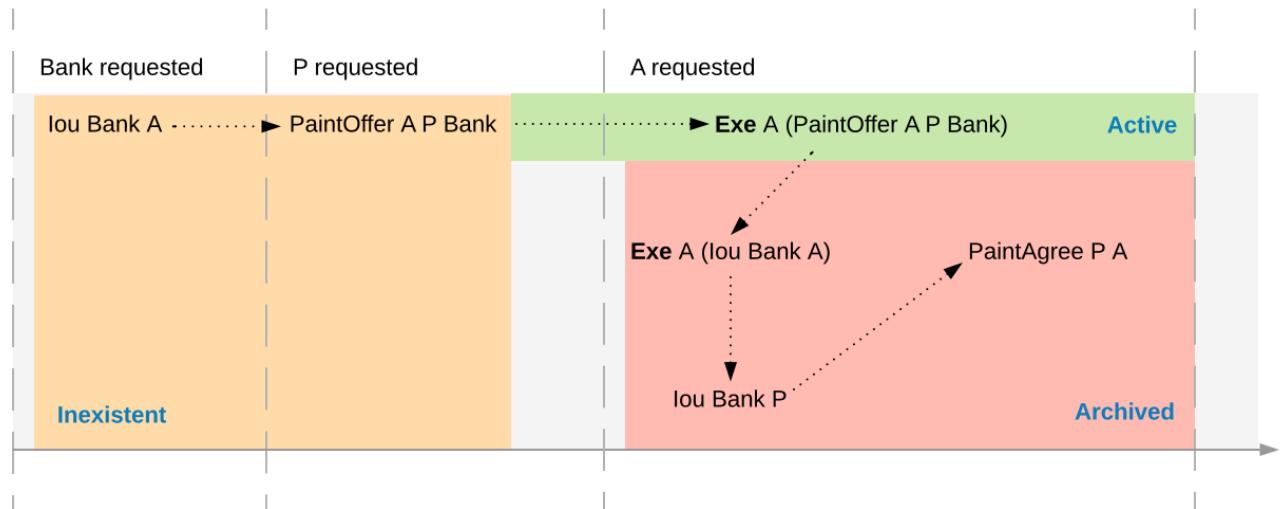


Fig. 6: Activeness of the *PaintOffer* contract

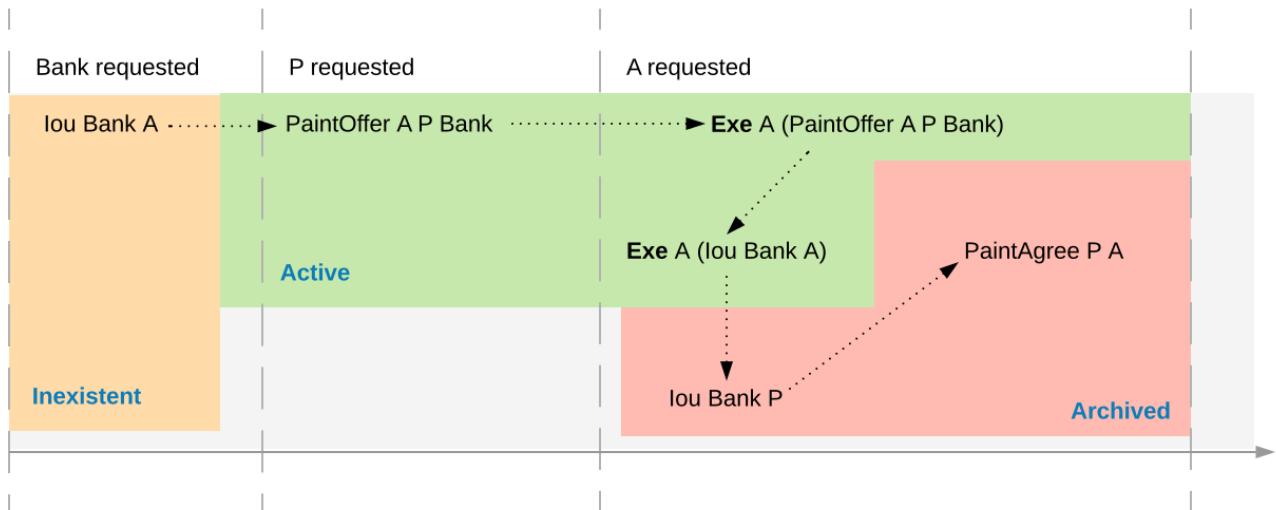
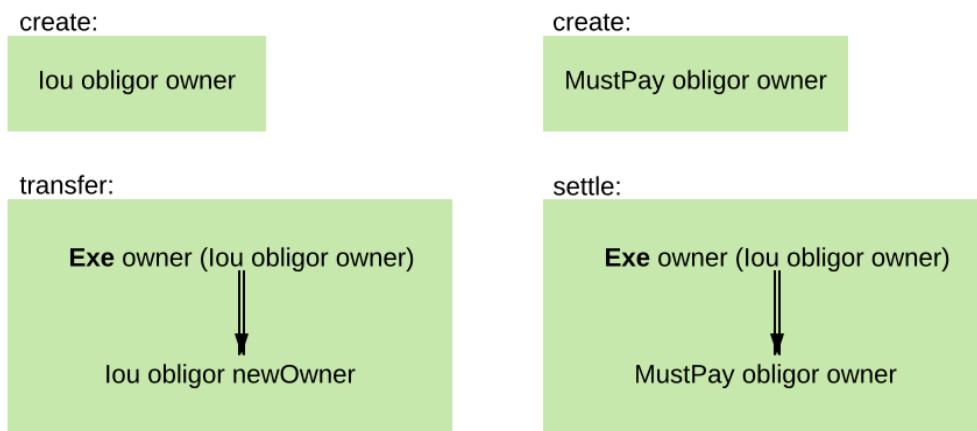


Fig. 7: Activeness of the *Iou Bank A* contract

### 5.1.2.3 Conformance

The conformance condition constrains the actions that may occur on the ledger. This is done by considering a **contract model M** (or a **model** for short), which specifies the set of all possible actions. A ledger is **conformant to M** (or conforms to M) if all top-level actions on the ledger are members of M. Like consistency, the notion of conformance does not depend on the requesters of a commit, so it can also be applied to transactions and lists of transactions.

For example, the set of allowed actions on IOU contracts could be described as follows.



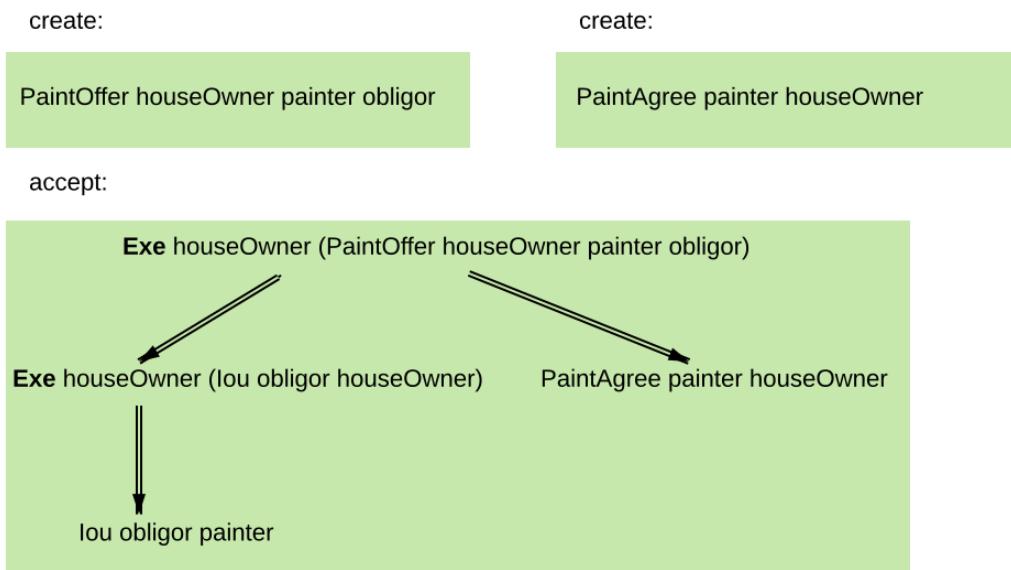
The boxes in the image are templates in the sense that the contract parameters in a box (such as obligor or owner) can be instantiated by arbitrary values of the appropriate type. To facilitate understanding, each box includes a label describing the intuitive purpose of the corresponding set of actions. As the image suggests, the transfer box imposes the constraint that the bank must remain the same both in the exercised IOU contract, and in the newly created IOU contract. However, the owner can change arbitrarily. In contrast, in the settle actions, both the bank and the owner must remain the same.

As indicated in the visualization, the contract model specifies the possible actors for each exercise action. For example, it specifies that owner has to be the actor of transfer. If the model contains an

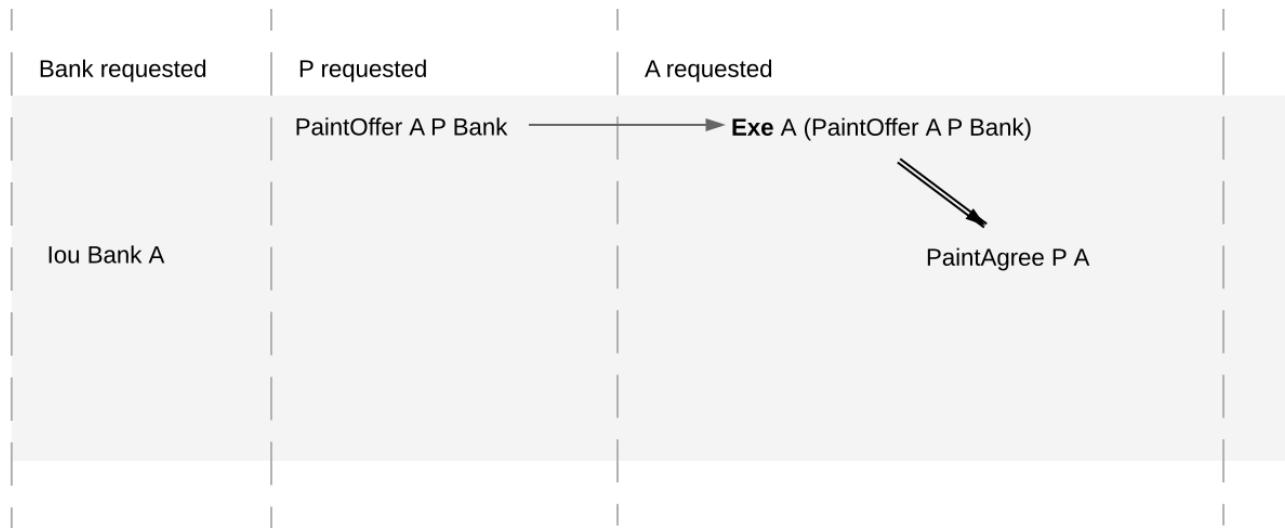
action on a contract  $c$  with  $p$  as an actor,  $p$  is said to be a **controller** of  $c$  in the model.

Of course, the restrictions on the relationship between the parameters can be arbitrary complex, and cannot conveniently be reproduced in this graphical representation. This is the role of DAML – it provides a much more convenient way of representing contract models.

To see the conformance criterion in action, assume that the contract model allows only the following actions on *PaintOffer* and *MustPaint* contracts.



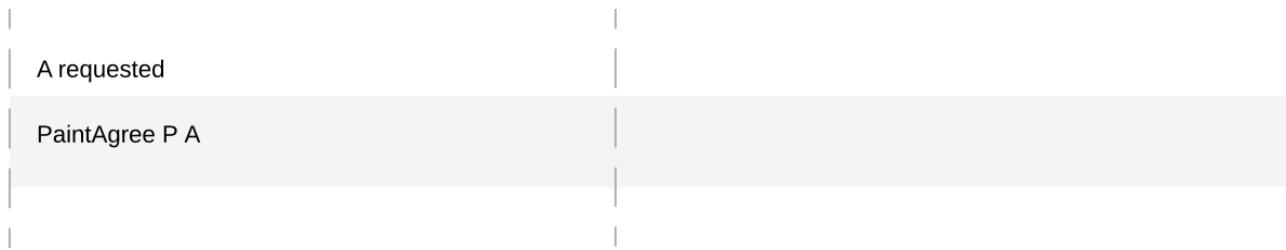
The problem with example where Alice changes the offer's outcome to avoid transferring the money now becomes apparent.



$A$ 's commit is not conformant to the contract model, as the model does not contain the top-level action she is trying to commit.

#### 5.1.2.4 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 and Agreements

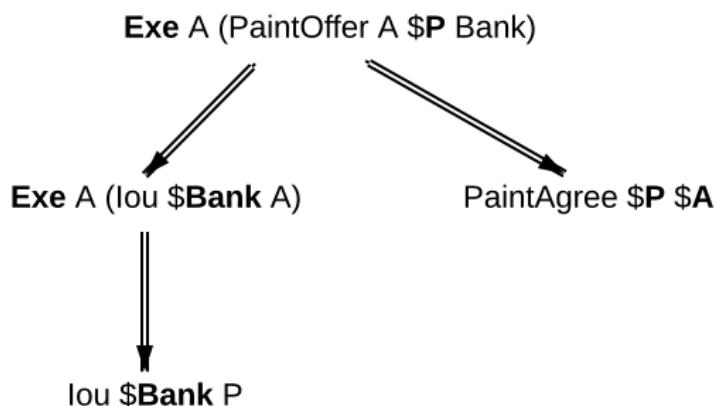
To capture these elements of real-world contracts, the **contract model** additionally specifies, for each contract in the system:

1. a non-empty set of **signatories**, the parties bound by the contract, and
2. an optional **agreement text** associated with the contract, specifying the off-ledger, real-world obligations of the signatories.

In the example, the contract model specifies that

1. a `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 contract has only the `painter` as the signatory, with no agreement text.
4. a `PaintAgree` `houseOwner` `painter` contract has both the `house owner` and the `painter` as signatories, with an agreement text requiring the painter to paint the house.

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. For example, annotating the paint offer acceptance action with signatories yields the image below.



## Authorization Rules

Signatories allow one to precisely state that the painter has an obligation. The imposed obligation is intuitively invalid because the painter did not agree to this obligation. In other words, the painter did not authorize the creation of the obligation.

In a DA ledger, a party can **authorize** a subaction of a commit in one of the two ways:

Every top-level action of the commit is authorized by all requesters of the commit.

Every consequence of an exercise action act on a contract  $c$  is authorized by all signatories of  $c$  and all actors of act.

The second authorization rule encodes the offer-acceptance pattern, which is a prerequisite for contract formation in contract law. The contract  $c$  is effectively an offer by its signatories who act as offerers. The exercise is an acceptance of the offer by the actors who are the offerees. The consequences of the exercise can be interpreted as the contract body so the authorization rules of DA ledgers closely model the rules for contract formation in contract law.

A commit is **well-authorized** if every subaction act of the commit is authorized by at least all of the **required authorizers** of act, where:

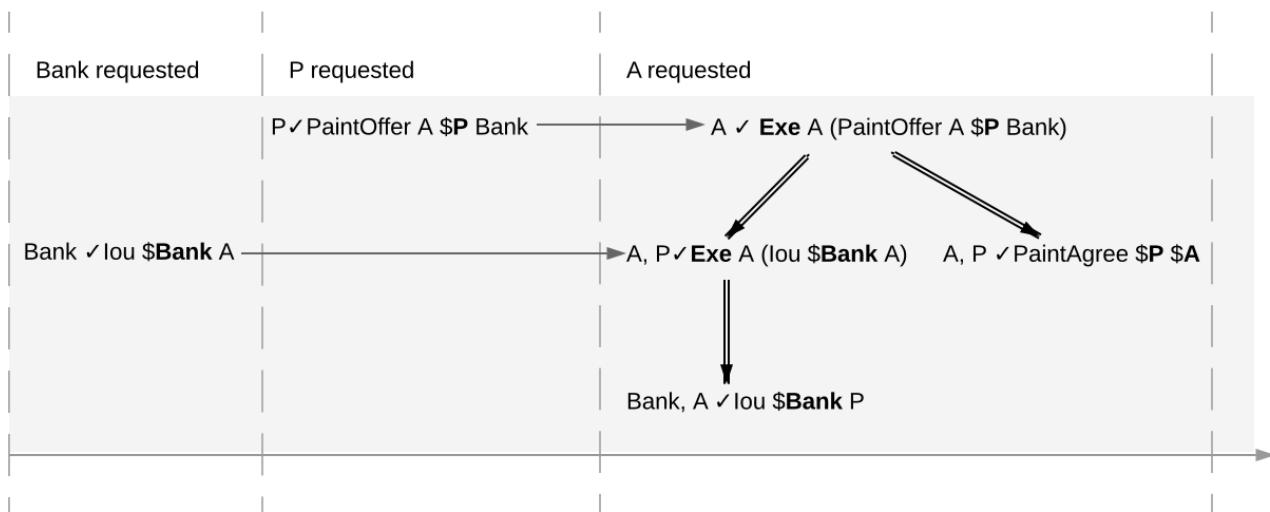
1. the required authorizers of a **Create** action on a contract  $c$  are the signatories of  $c$ .
2. the required authorizers of an **Exercise** or a **Fetch** action are its actors.

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.

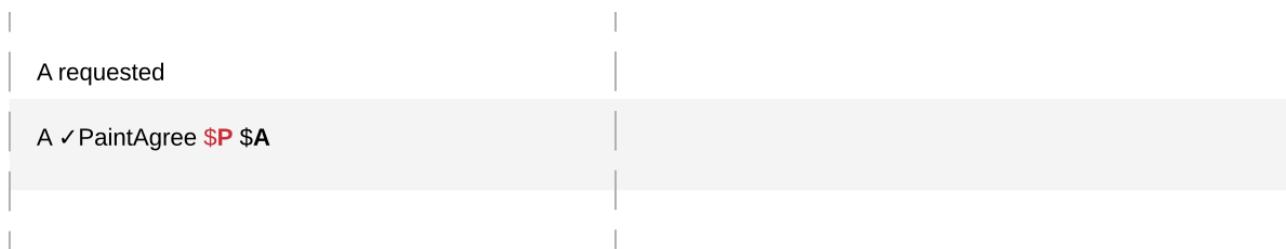
In the visualizations below,  $\Pi \checkmark$  act denotes that the parties  $\Pi$  authorize the action act. The resulting authorizations are shown below.



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.

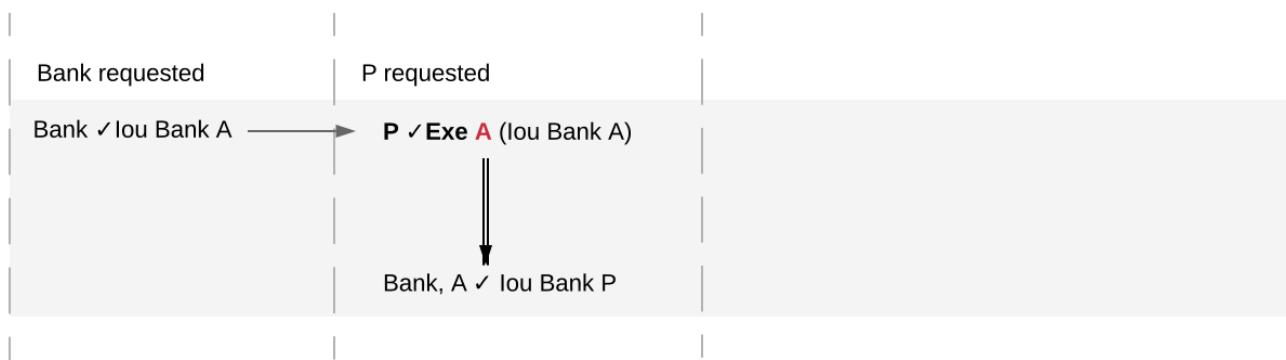
The third commit is more complicated. First, Alice authorizes the exercise on the paint offer by requesting it. She is the only actor on this exercise, so this complies with the authorization requirement. Since the painter is the signatory of the paint offer, and Alice the actor of the exercise, they 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.

### 5.1.2.5 Valid Ledgers, Obligations, Offers and Rights

DA ledgers are designed to mimic real-world interactions between parties, which are governed by contract law. The validity conditions on the ledgers, and the information contained in contract models have several subtle links to the concepts of the contract law that are worth pointing out.

First, in addition to the explicit off-ledger obligations specified in the agreement text, contracts also specify implicit **on-ledger obligations**, which result from consequences of the exercises on con-

tracts. For example, the *PaintOffer* contains an on-ledger obligation for A to transfer her IOU in case she accepts the offer. Agreement texts are therefore only necessary to specify obligations that are not already modeled as permissible actions on the ledger. For example, P's obligation to paint the house cannot be sensibly modeled on the ledger, and must thus be specified by the agreement text.

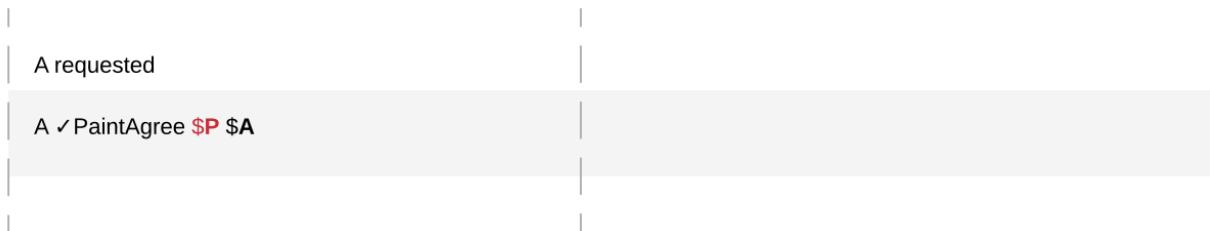
Second, every contract on a DA ledger can simultaneously model both:

- a real-world offer, whose consequences (both on- and off-ledger) are specified by the **Exercise** actions on the contract allowed by the contract model, and
- a real-world contract proper, specified through the contract's (optional) agreement text.

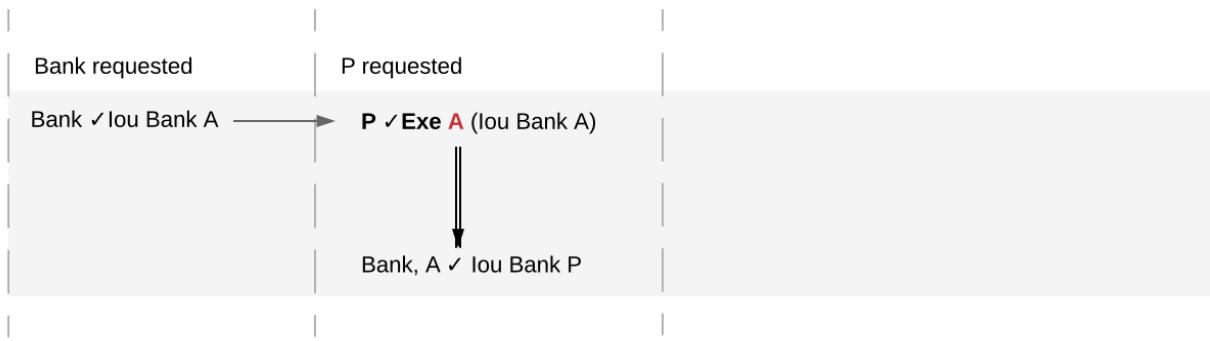
Third, in DA ledgers, as in the real world, one person's rights are another person's obligations. For example, A's right to accept the *PaintOffer* is P's obligation to paint her house in case she accepts. In DA ledgers, having (on-ledger) rights on a contract means being a controller of the contract (by the contract model).

Finally, validity conditions ensure three important properties of the DA ledger model, that mimic the contract law.

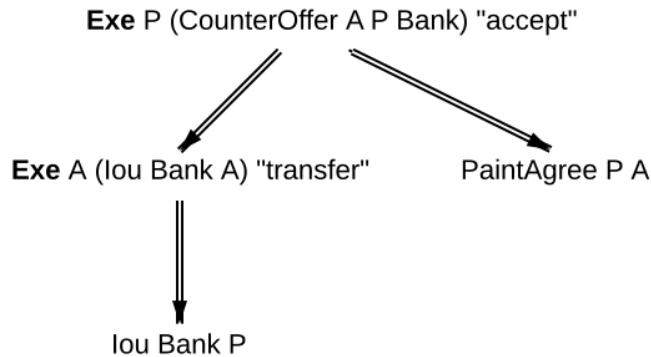
1. **Obligations need consent.** DA ledgers follow the offer-acceptance pattern of the contract law, and thus ensures that all ledger contracts are formed voluntarily. For example, the following ledger is not valid.



2. **Consent is needed to take away on-ledger rights.** As only **Exercise** actions consume contracts, the rights cannot be taken away from the actors; the contract model specifies exactly who the actors are, and the authorization rules require them to approve the contract consumption. In the examples, Alice had the right to transfer her IOUs; painter's attempt to take that right away from her, by performing a transfer himself, was not valid.



Parties can still **delegate** their rights to other parties. For example, assume that Alice, instead of accepting painter's offer, decides to make him a counteroffer instead. The painter can then accept this counteroffer, with the consequences as before:



Here, by creating the `CounterOffer` contract, Alice delegates her right to transfer the IOU contract to the painter. In case of delegation, prior to submission, the requester must get informed about the contracts that are part of the requested transaction, but where the requester is neither a signatory nor a controller. In the example above, the painter must learn about the existence of the IOU for Alice before he can request the acceptance of the `CounterOffer`. The concepts of observers and divulgence, introduced in the next section, enable such scenarios.

3. **On-ledger obligations cannot be unilaterally escaped.** Once an obligation is recorded on a DA ledger, it can only be removed in accordance with the contract model. For example, assuming the IOU contract model shown earlier, if the ledger records the creation of a `MustPay` contract, the bank cannot later simply record an action that consumes this contract:

		Bank commits
...		Exe Bank (MustPay Bank A)
...		
MustPay Bank A		

That is, this ledger is invalid, as the action above is not conformant to the contract model.

### 5.1.3 Privacy

The previous sections have addressed two out of three questions posed in the introduction: what the ledger looks like, and who may request which changes. This section addresses the last one, who sees which changes and data. That is, it explains the privacy model for DA ledgers.

The privacy model of the DA platform is based on a **need-to-know basis**, and provides privacy **on the level of subtransactions**. Namely, a party learns only those parts of ledger changes that affect contracts in which the party has a stake, and the consequences of those changes.

To make this more precise, a stakeholder concept is needed.

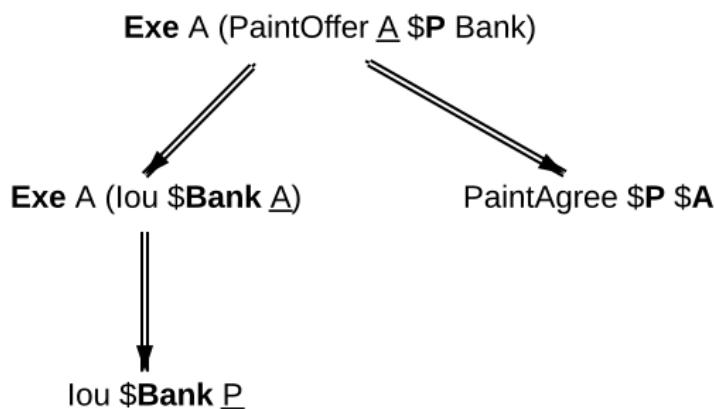
#### 5.1.3.1 Contract Observers and Stakeholders

Intuitively, at least anyone with an explicit right or obligation on a contract should be considered a stakeholder. That means all signatories and controllers of a contract action have a stake. In addition, it makes sense to define general **observers**, parties who have the right to see the contract, but are not bound by the contract and can not change the contract.

For example, in the offer process, the *PaintOffer*, Alice is able to accept the offer so she is a controller. As a stakeholder, she is made aware of the offer's existence.

Signatories and controllers 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, a controller, or an observer on the contract.

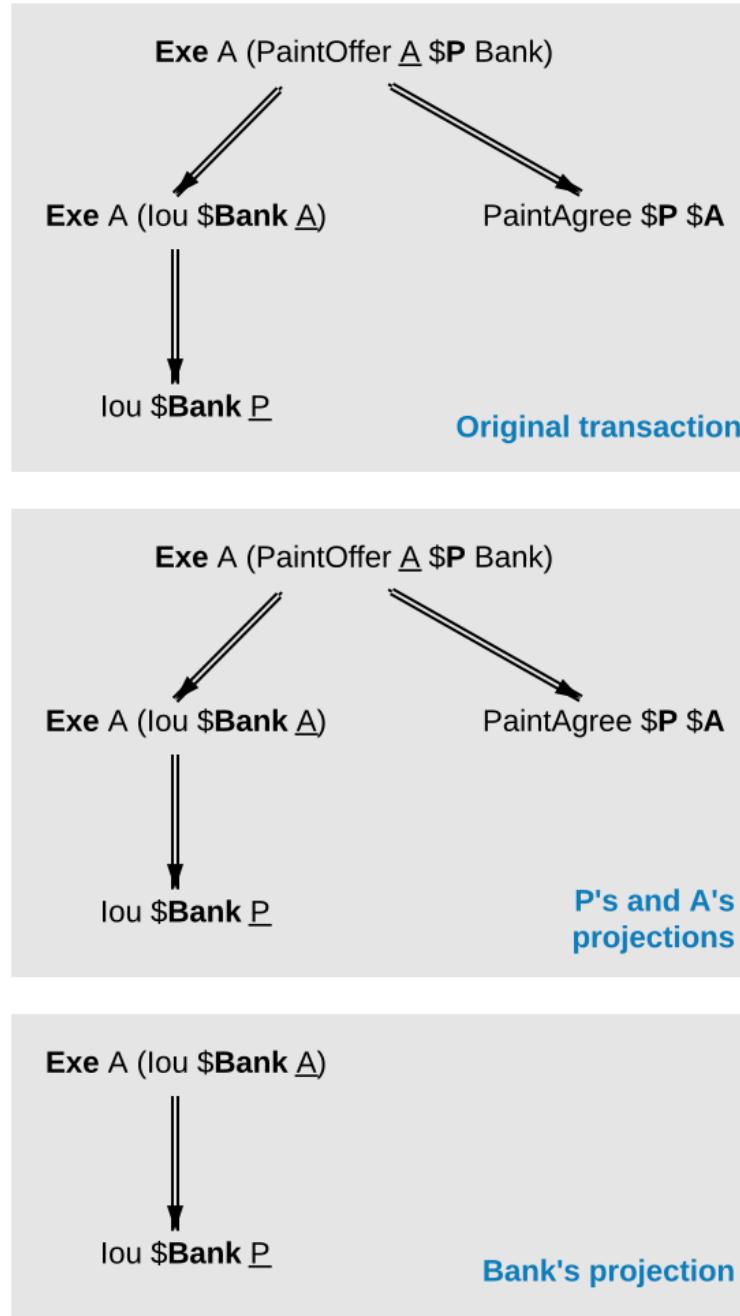
In the graphical representation of the paint offer acceptance below, controllers, which are not also signatories, are indicated by an underline.



### 5.1.3.2 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.

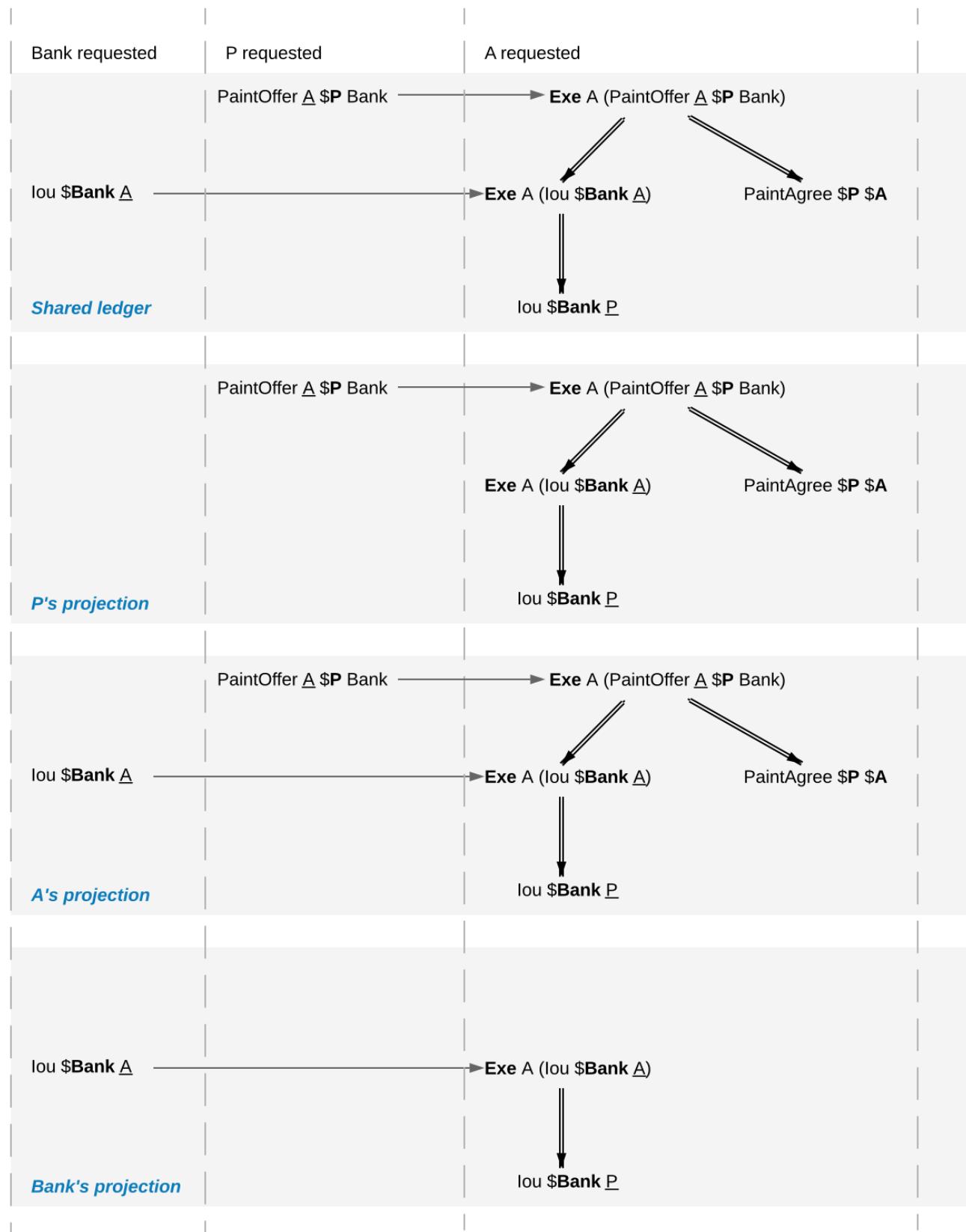
As a design choice, the DA Platform shows to observers on a contract only the *state changing* actions on the contract. More precisely, **Fetch** and non-consuming **Exercise** actions are not shown to the observers - except when they are the actors of these actions. This motivates the following definition: a party  $p$  is an **informee** of an action  $A$  if one of the following holds:

- A is a **Create** on a contract  $c$  and  $p$  is a stakeholder of  $c$ .
- A is a consuming **Exercise** on a contract  $c$  and  $p$  is a stakeholder of  $c$ .
- 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  $A$ .

Then, we can formally define the **projection** of a transaction  $tx = act_1, , act_n$  for a party  $p$  is the sub-transaction obtained by doing the following for each action  $act_i$ :

1. If  $p$  is an informee of  $act_i$ , keep  $act_i$  as-is.
2. Else, if  $act_i$  has consequences, replace  $act_i$  by the projection (for  $p$ ) of its consequences, which might be empty.
3. Else, drop  $act_i$ .

Finally, the **projection of a ledger**  $l$  for a party  $p$  is a list of transactions obtained by first projecting the transaction of each commit in  $l$  for  $p$ , and then removing all empty transactions from the result. Note that the projection of a ledger is not a ledger, but a list of transactions. Projecting the ledger of our complete paint offer example yields the following projections for each party:



Examine each party's projection in turn:

1. The painter does not see any part of the first commit, as he is not a stakeholder of the *iou Bank A* contract. Thus, this transaction is not present in the projection for the painter at all. However, the painter is a stakeholder in the *PaintOffer*, so he sees both the creation and the exercise of

this contract (again, recall that all consequences of an exercise action are a part of the action itself).

2. Alice is a stakeholder in both the *lou* 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, *lou* 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, and is consistent for all contracts on which p is a stakeholder. In other words, p is never a stakeholder on any input contracts of its projection. Furthermore, if the contract model is **subaction-closed**, which means that for every action act in the model, all subactions of act are also in the model, then the projection is guaranteed to be conformant. As we will see shortly, DAML-based contract models are conformant. Lastly, as projections carry no information about the requesters, we cannot talk about authorization on the level of projections.

### 5.1.3.3 Divulgence: When Non-Stakeholders See Contracts

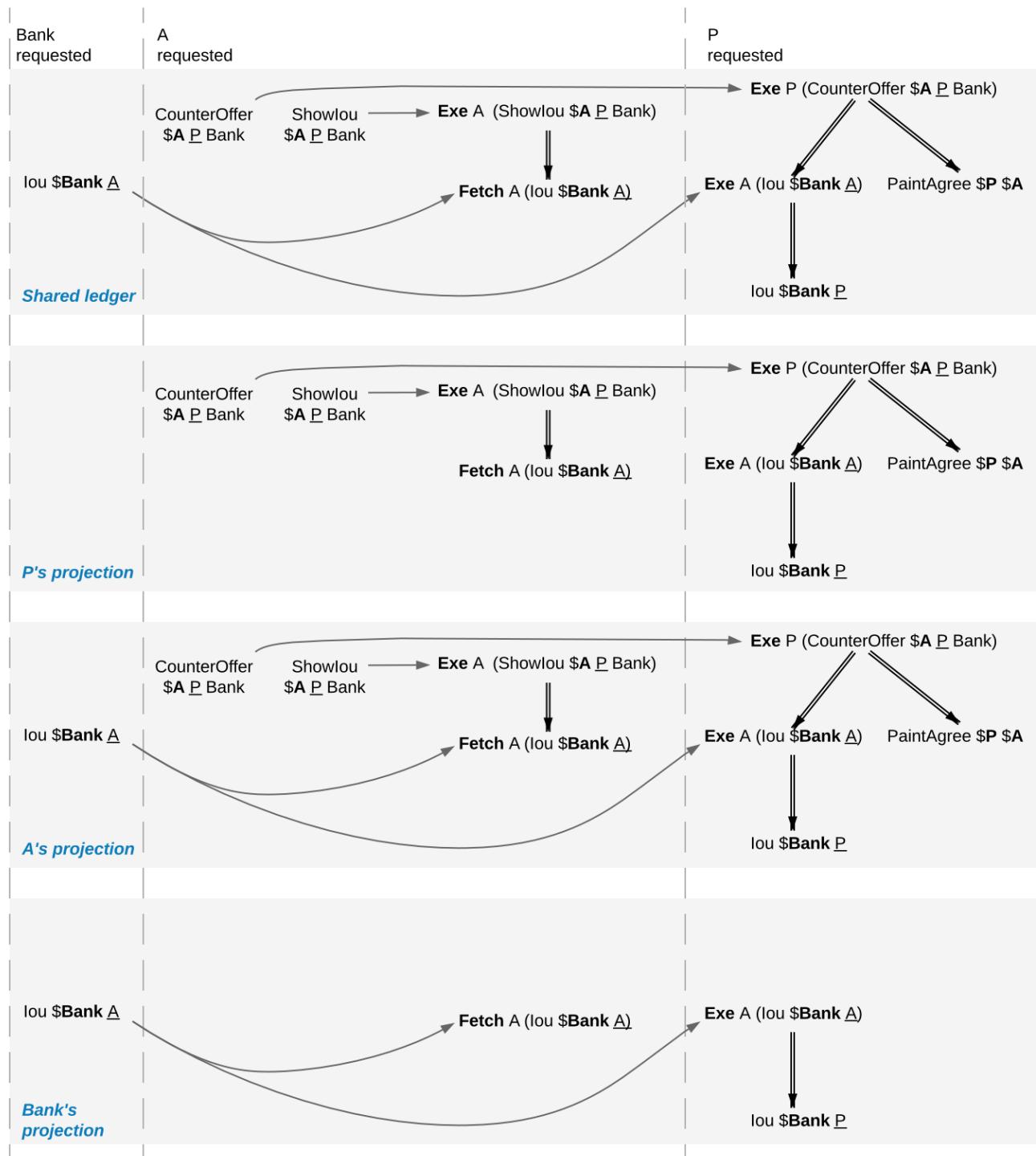
The guiding principle for the privacy model of DA ledgers is that contracts should only be shown to their stakeholders. However, ledger projections can cause contracts to become visible to other parties as well.

In the example of *ledger projections of the paint offer*, the exercise on the PaintOffer is visible both the painter and to Alice. As a consequence, the exercise on the *lou* Bank A is visible to the painter, and the creation of *lou* Bank P is visible to Alice. As actions also contain the contracts they act on, *lou* Bank A was thus shown to the painter and *lou* Bank P was shown to Alice.

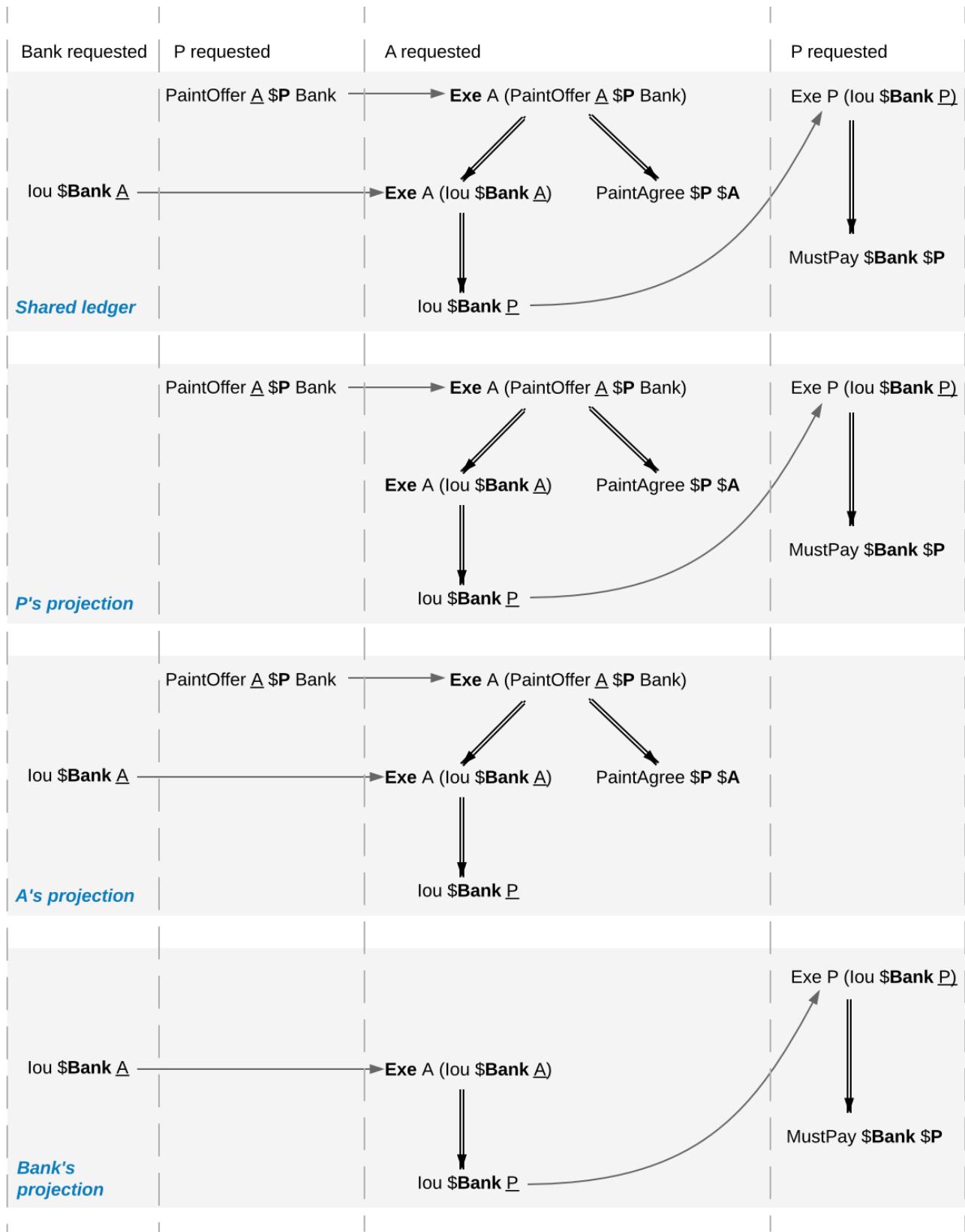
Showing contracts to non-stakeholders through ledger projections is called **divulgence**. Divulgence is a deliberate choice in the design of DA ledgers. In the paint offer example, the only proper way to accept the offer is to transfer the money from Alice to the painter. Conceptually, at the instant where the offer is accepted, its stakeholders also gain a temporary stake in the actions on the two *lou* 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 **Showlou** 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 `lou Bank P` contract, she does not see the settlement action. Thus, she does not know whether the contract is still active at any point after its creation. Similarly, in the previous example with the counteroffer, Alice could spend the IOU that she showed to the painter by the time the painter attempts to accept her counteroffer. In this case, the painter's transaction

could not be added to the ledger, as it would result in a double spend and violate validity. But the painter has no way to predict whether his acceptance can be added to the ledger or not.

#### 5.1.4 DAML: Defining Contract Models Compactly

As described in preceding sections, both the integrity and privacy notions depend on a contract model, and such a model must specify:

1. a set of allowed actions on the contracts, and
2. the signatories, controllers, observers, and agreement text associated with each contract

The sets of allowed actions can in general be infinite. For instance, the actions in the IOU contract model considered earlier can be instantiated for an arbitrary obligor and an arbitrary owner. As enumerating all possible actions from an infinite set is infeasible, a more compact way of representing models is needed.

DAML provides exactly that: a compact representation of a contract model. Intuitively, the allowed actions are:

1. **Create** actions on all instances of DAML templates such that the template arguments satisfy the ensure clause of the template
2. **Exercise** actions on a contract instance corresponding to DAML choices on that template, with given choice arguments, such that:
  1. The actors match the **controllers** of the choice.
  2. The exercise kind matches.
  3. All assertions in the update block hold for the given choice arguments.
  4. Create, exercise and fetch statements in the DAML update block are represented as create and exercise actions in the consequences of the exercise action.
3. **Fetch** actions on a contract instance corresponding to a fetch of that instance inside of an update block. The actors must be a non-empty subset of the contract stakeholders. The actors are determined dynamically as follows: if the fetch appears in an update block of a choice *ch* on a contract *c1*, and the fetched contract ID resolves to a contract *c2*, then the actors are defined as the intersection of (1) the signatories of *c1* union the controllers of *ch* with (2) the signatories of *c2*.

An instance of a DAML template, that is, a **DAML contract** or **contract instance**, is a triple of:

1. a contract identifier
2. the template identifier
3. the template arguments

The signatories, controllers, and observers of a DAML contract are derived from the template arguments and the explicit signatory, controller, and observer annotations on the contract template.

For example, the following DAML template exactly describes the contract model of a simple IOU with a unit amount, shown earlier.

```
template MustPay with
  obligor : Party
  owner : Party
where
```

(continues on next page)

(continued from previous page)

```
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
```

Furthermore, the template identifiers of DAML contracts are created through a content-addressing scheme. This means every DAML contract is self-describing in a sense: it constrains its stakeholder annotations and all DAML-conformant actions on itself. As a consequence, one can talk about the DAML contract model, as a single contract model encoding all possible instances of all possible DAML templates. This model is subaction-closed; all exercise and create actions done within an update block are also always permissible as top-level actions.

# Chapter 6

## Examples

### 6.1 DAML examples

We have plenty of example code, both of DAML and of applications around DAML, on our [public GitHub organization](#).

[Bond trading example](#): DAML code and automation using the Java bindings

[Collateral management example](#): DAML code

[Repurchase agreement example](#): DAML code and automation using the Java bindings

[Upgrading DAML templates example](#): DAML code

[Java bindings tutorial](#): three examples using the Java bindings with a very simple DAML model

[Node.js tutorial](#): step-by-step running through using the experimental Node.js bindings

# Chapter 7

## Experimental Features

### 7.1 WARNING

The tools described in this section are actively being designed and are subject to breaking changes or removal. When we become more confident in their designs, we will introduce them as standard components in the SDK.

#### 7.1.1 Node.js bindings

##### 7.1.1.1 Getting started

###### 1. Set up NPM

The Node.js bindings are published on NPM.

To set up NPM:

1. [Login to Bintray](#)
2. go to the [Digital Asset NPM repository home page on Bintray](#)
3. click on the Set me up! button
4. follow the instructions for scoped packages with scope `da`

###### 2. Start a new project

Use `npm init` to create a new project.

###### 3. Install the bindings

Use the NPM command line interface:

```
npm install @da/daml-ledger
```

## 4. Start coding

To guide you through using the Node.js bindings, we provide a tutorial template and a [tutorial](#).

### 7.1.1.2 Tutorial

This tutorial guides you through a series of steps to write a simple application.

The purpose is to learn the basics of how to use the Node.js bindings.

The task is to build an application able to send and receive ping messages.

The focus is not on the complexity of the model, but rather on how to use the bindings to interact with the ledger.

#### Create the project

There is a skeleton application called `tutorial-nodejs` that you can get from the public repository [github.com/digital-asset/ex-tutorial-nodejs](https://github.com/digital-asset/ex-tutorial-nodejs). To set it up, clone the repo:

```
$ git clone git@github.com:digital-asset/ex-tutorial-nodejs.git  
$ cd ex-tutorial-nodejs
```

The repo includes `daml/PingPong.daml`, which is the source for a DAML module with two templates: Ping and Pong. The app uses these.

#### Run the sandbox

Use the sandbox to run and test your application.

Now that you created the project and you're in its directory, start the sandbox by running:

```
da sandbox
```

Starting the sandbox automatically compiles `PingPong.daml` and loads it.

#### Run the skeleton app

You are now set to write your own application. The template includes a skeleton app that connects to a running ledger and quits.

1. Install the dependencies for your package (including the bindings):

```
npm install
```

2. Start the application:

```
npm start
```

3. Verify the output is correct

```
hello from <LEDGER_ID>
```

## Understand the skeleton

The code for the script you just ran is `index.js`.

Let's go through the skeleton part by part to understand what's going on:

```
const da = require('@da/daml-ledger').data;  
  
const uuidv4 = require('uuid/v4');
```

The first line loads the bindings and allows you to refer to them through the `da` object.

The second one introduces a dependency that is going to be later used to generate unique identifiers; no need to worry about it now.

```
let [, , host, port] = process.argv;  
  
host = host || 'localhost';  
port = port || 7600;
```

These lines read the command-line arguments and provide some sensible defaults.

Now to the juicy part:

```
da.DamlLedgerClient.connect({ host: host, port: port }, (error, client) =>  
  ↵ {  
    if (error) throw error;  
    console.log('hello from', client.ledgerId);  
  });
```

Here the application connects to the ledger with the `DamlLedgerClient.connect` method.

It accepts two arguments:

- an object with the connection options
- a callback to be invoked when the connection either fails or succeeds

The connection options require you to pass the `host` and `port` of the ledger instance you are connecting to.

The callback follows the common pattern in Node.js of being invoked with two arguments: the first is an error in case of failure while the latter is the response in case of success.

In this case in particular, the response in case of success is a `client` object that can be used to communicate with the ledger.

The skeleton application just prints the greeting message with the ledger identifier and quits.

## Retrieve the package identifiers

Now that the sandbox is running, the `PingPong.daml` file has been compiled and the module loaded onto the ledger.

In order for you to refer to the templates therein you need its package identifier.

This template includes a script that connects to a running ledger instance and downloads the package identifiers for the templates.

Run it now:

```
npm run fetch-template-ids
```

If the program ran successfully, the project root now contains the `template-ids.json` file.

It's time to write some code to verify that you're good to go. Open the `index.js` file and edit it.

First of all, right after the first `require` statement, add a new one to load the `template-ids.json` file that has just been created.

```
const da = require('@da/daml-ledger').data;
const templateIds = require('./template-ids.json');
```

Right beneath that line, initialize two constants to hold the `Ping` and `Pong` template identifiers:

```
const PING = templateIds['PingPong.Ping'];
const PONG = templateIds['PingPong.Pong'];
```

Finally print the template identifiers:

```
da.DamlLedgerClient.connect({ host: host, port: port }, (error, client) =>
  ↵ {
    if (error) throw error;
    console.log('hello from', client.ledgerId);
    console.log('Ping', PING);
    console.log('Pong', PONG);
  });
});
```

Run the application again (`npm start`) to see an output like the following:

```
hello from sandbox-3957952d-f475-4d2f-be89-245a0799d2c0
Ping { packageId:
  '5976641aeea761fa8946ea004b318a74d869ee305fafcdc6bf98d31fa354304d',
  name: 'PingPong.Ping' }
Pong { packageId:
  '5976641aeea761fa8946ea004b318a74d869ee305fafcdc6bf98d31fa354304d',
  name: 'PingPong.Pong' }
```

## The PingPong module

Before moving on to the implementation of the application, have a look at `daml/PingPong.daml` to understand the module the app uses.

`Ping` and `Pong` are almost identical. Looking at them in detail:

both have a sender signatory and a receiver observer  
the receiver of a `Ping` can exercise the `ReplyPong` choice, creating a `Pong` contract with swapped sender and receiver

symmetrically, the receiver of a Pong contract can exercise the ReplyPing choice, creating a Ping contract with swapped parties

Note that the contracts carry a counter: when the counter reaches 3, no new contract is created and the exchange stops.

## Pass the parties as parameters

Everything's now ready to start working. Edit the index.js file.

Each contract has a sender and a receiver, so your application needs to establish it.

Read those from the command line by editing the part where the arguments are read as follows:

```
let [, , sender, receiver, host, port] = process.argv;
host = host || 'localhost';
port = port || 7600;
if (!sender || !receiver) {
  console.log('Missing sender and/or receiver arguments, exiting.');
  process.exit(-1);
}
```

Try to run it without arguments (or with just one) to see the error popping up.

Try to run it with both arguments to see the application working just as it did before.

## Create a contract

To kickstart the exchange between two parties you have to first make one party send the initial ping to the other.

To do this you need to create a Ping contract.

This requires you to submit a command to the ledger. For this, use the CommandService.

The client object returned by the DamlLedgerClient.connect method contains a reference to all services exposed by the ledger, including the CommandService.

First of all, the following is the request for the CommandService. Have a look at it:

```
const request = {
  commands: [
    applicationId: 'PingPongApp',
    workflowId: `Ping-${sender}`,
    commandId: uuidv4(),
    ledgerEffectiveTime: { seconds: 0, nanoseconds: 0 },
    maximumRecordTime: { seconds: 5, nanoseconds: 0 },
    party: sender,
    list: [
      {
        create: {
          templateId: PING,
          arguments: {
            ...
```

(continues on next page)

(continued from previous page)

```

        fields: {
            sender: { party: sender },
            receiver: { party: receiver },
            count: { int64: 0 }
        }
    }
}
]
}
};


```

This object represents the submission of a set of commands to be applied atomically. Let's see what each bit of it means:

- applicationId** the name of your application
- commandId** a unique identifier for the set of submitted commands
- workflowId** an (optional) identifier you can use to group together commands pertaining to one of your workflows
- ledgerEffectiveTime** the time at which the set of submitted commands are applied; normally the client's current epoch time, but, since the sandbox (by default) runs with a static time fixed at epoch 0, use this value
- maximumRecordTime** the time at which the command is considered expired if it's not been applied yet; the difference with the `maximumRecordTime` is the time-to-live (TTL) of the command
- party** who's submitting the command

Finally, `list` contains all the commands to be applied. In this case, it submits a `create` command.

Have a look at the only command:

- templateId** the identifier of the template of the contract you wish to create (`Ping`)
- arguments** an object containing the `fields` necessary to create the contract

The keys of the `fields` object are the template parameter names as they appear on `daml/PingPong.daml`, while the values are a pair with the type and the value being passed (in this case two parties).

The request can now be passed to the `CommandService` as follows:

```

client.commandClient.submitAndWait(request, (error, _) => {
    if (error) throw error;
    console.log(`Created Ping contract from ${sender} to ${receiver}.`);
});

```

This is already a sizeable chunk of code that performs a clearly defined task. Within the body of the `connect` callback, wrap the code from this section in a function called `createFirstPing` and call it.

The code should now look like the following:

```

da.DamlLedgerClient.connect({ host: host, port: port }, (error, client) =>
    ↵{

```

(continues on next page)

(continued from previous page)

```
if (error) throw error;

createFirstPing();

function createFirstPing() {
  const request = {
    commands: [
      applicationId: 'PingPongApp',
      workflowId: `Ping-${sender}`,
      commandId: uuidv4(),
      ledgerEffectiveTime: { seconds: 0, nanoseconds: 0 },
      maximumRecordTime: { seconds: 5, nanoseconds: 0 },
      party: sender,
      list: [
        {
          create: {
            templateId: PING,
            arguments: {
              fields: {
                sender: { party: sender },
                receiver: { party: receiver },
                count: { int64: 0 }
              }
            }
          }
        }
      ]
    }
  };
  client.commandClient.submitAndWait(request, (error, _) => {
    if (error) throw error;
    console.log(`Created Ping contract from ${sender} to $` +
    ↵{receiver}.`);
  });
}

});
```

Time to test your application. Run it like this:

npm start Alice Bob

You should see the following output:

Created Ping contract from Alice to Bob.

Your application now successfully creates a Ping contract on the ledger, congratulations!

## Read the transactions

Now that the application can create a contract to send a `ping`, it must also be able to listen to `pongs` on the ledger so that it can react to those.

The `TransactionService` exposes the functionality to read transactions from the ledger via the `getTransactions` method.

This method takes the following request:

```
const filtersByParty = {};
filtersByParty[sender] = { inclusive: { templateIds: [PING, PONG] } };
const request = {
    begin: { boundary: da.LedgerOffset.Boundary.END },
    filter: { filtersByParty: filtersByParty }
};
```

Have a look at the request:

- begin** the offset at which you'll start reading transactions from the ledger. In this case you want to listen starting from the latest one (represented by the constant `da.LedgerOffset.Boundary.END`)
- end** the optional offset at which you want the reads to end - if absent (as in this case) the application keeps listening to incoming transactions
- filter** represents which contracts you want the ledger to show you: in this case you are asking for the transactions visible to sender containing contracts whose `templateId` matches either `PING` or `PONG`.

When the `getTransactions` method is invoked with this request the application listens to the latest transactions coming to the ledger.

The output of this method is a Node.js stream. As such, you can register callbacks on the '`data`' and '`error`' events.

The following code prints the incoming transaction and quits in case of '`error`'.

```
const transactions = client.transactionClient.getTransactions(request);
console.log(`\$ {sender} starts reading transactions.`);
transactions.on('data', response => {
    for (const transaction of response.transactions) {
        console.log('Transaction read:', transaction.transactionId);
    }
});
transactions.on('error', error => {
    console.error(`\$ {sender} encountered an error while processing
    ↵transactions!`);
    console.error(error);
    process.exit(-1);
});
```

---

**Note:** If your request specified an `end`, it would most probably make sense to register an '`end`' event callback on the stream as well.

Again, this code represents a sizeable chunk of code with a clearly defined purpose.

Wrap this code in a new function called `listenForTransactions`, place it within the `connect` callback and call `listenForTransactions` right before you call `createFirstPing`.

When you are done, your code should look like the following:

```
da.DamlLedgerClient.connect({ host: host, port: port }, (error, client) =>
  ↪{
    if (error) throw error;

    listenForTransactions();
    createFirstPing();

    function createFirstPing() {
      const request = {
        commands: [
          applicationId: 'PingPongApp',
          workflowId: `Ping-${sender}`,
          commandId: uuidv4(),
          ledgerEffectiveTime: { seconds: 0, nanoseconds: 0 },
          maximumRecordTime: { seconds: 5, nanoseconds: 0 },
          party: sender,
          list: [
            {
              create: {
                templateId: PING,
                arguments: {
                  fields: {
                    sender: { party: sender },
                    receiver: { party: receiver },
                    count: { int64: 0 }
                  }
                }
              }
            }
          ]
        }
      };
      client.commandClient.submitAndWait(request, error => {
        if (error) throw error;
        console.log(`Created Ping contract from ${sender} to ${receiver}`);
      });
    }

    function listenForTransactions() {
      console.log(`${sender} starts reading transactions.`);
      const filtersByParty = {};
      filtersByParty[sender] = { inclusive: { templateIds: [PING, PONG] } }
    };
    const request = {
```

(continues on next page)

(continued from previous page)

```

begin: { boundary: da.LedgerOffset.Boundary.END },
filter: { filtersByParty: filtersByParty }
};

const transactions = client.transactionClient.
getTransactions(request);
transactions.on('data', response => {
    for (const transaction of response.transactions) {
        console.log('Transaction read:', transaction.
transactionId);
    }
});
transactions.on('error', error => {
    console.error(`\$ {sender} encountered an error while processing
transactions!`);
    console.error(error);
    process.exit(-1);
});
}

);

```

Your application now should:

- start listening to pings and pong visible to the sender
- create the first ping
- receive the ping it created and print its transaction identifier

If you now run

```
npm start Alice Bob
```

You should see an output like the following:

```

Alice starts reading transactions.
Created Ping contract from Alice to Bob.
Transaction read: 1

```

Your application is now able to create contracts and listen to transactions on the ledger. Very good! You can now hit CTRL-C to quit the application.

### Exercise a choice

The last piece of functionality you need consists of reacting to pings and pong that you read from the ledger, represented by the creation of contracts.

For this, use again the submitAndWait method.

In particular, make your program exercise a choice: ReplyPing when you receive a Pong and vice versa.

You need to react to events in transactions as they are received in the listenForTransactions function.

The transaction object whose transactionId you printed so far contains an array of event objects, each representing an archived or created event on a contract.

What you want to do is loop through the events in the transaction and extract the receiver and count fields from created events.

You then want to decide which reply to give (either ReplyPing or ReplyPong) based on the contract that has been read.

For each created event, you want to send a command that reacts to it, specifying that you want to either exercise the ReplyPing choice of a Pong contract or vice versa.

The following snippet of code does precisely this.

```
const reactions = [];
for (const event of events) {
    const { receiver: { party: receiver }, count: { int64: count } } =
        event.arguments.fields;
    if (receiver === sender) {
        const templateId = event.templateId;
        const contractId = event.contractId;
        const reaction = templateId.name == PING.name ? 'ReplyPong' :
            'ReplyPing';
        console.log(`$ ${sender} (workflow ${workflowId}): ${reaction} at
${count}`);
        reactions.push({
            exercise: {
                templateId: templateId,
                contractId: contractId,
                choice: reaction,
                argument: { record: { fields: {} } }
            }
        });
    }
}
```

You can now use the submitAndWait command to send the reactions to the ledger.

```
if (reactions.length > 0) {
    const request = {
        commands: {
            applicationId: 'PingPongApp',
            workflowId: workflowId,
            commandId: uidv4(),
            ledgerEffectiveTime: { seconds: 0, nanoseconds: 0 },
            maximumRecordTime: { seconds: 5, nanoseconds: 0 },
            party: sender,
            list: reactions
        }
    }
    client.commandClient.submitAndWait(request, error => {
        if (error) throw error;
    });
}
```

(continues on next page)

(continued from previous page)

}

Wrap this code into a new function `react` that takes a `workflowId` and an `events` array with the created events. Then edit the `listenForTransactions` function to:

- accept one parameter called `callback`
- instead of printing the transaction identifier, for each transaction
  - push the created events to an array
  - pass that array to the `callback` (along with the workflow identifier)

Finally, pass the `react` function as a parameter to the only call of `listenForTransactions`.

Your code should now look like the following:

```
da.DamlLedgerClient.connect({ host: host, port: port }, (error, client) =>
  ↪{
    if (error) throw error;

    listenForTransactions(react);
    createFirstPing();

    function createFirstPing() {
      const request = {
        commands: {
          applicationId: 'PingPongApp',
          workflowId: `Ping-${sender}`,
          commandId: uuidv4(),
          ledgerEffectiveTime: { seconds: 0, nanoseconds: 0 },
          maximumRecordTime: { seconds: 5, nanoseconds: 0 },
          party: sender,
          list: [
            {
              create: {
                templateId: PING,
                arguments: {
                  fields: {
                    sender: { party: sender },
                    receiver: { party: receiver },
                    count: { int64: 0 }
                  }
                }
              }
            }
          ]
        }
      };
      client.commandClient.submitAndWait(request, error => {
        if (error) throw error;
        console.log(`Created Ping contract from ${sender} to $` ↪{receiver}.`);
      });
    }
  }
);
```

(continues on next page)

(continued from previous page)

```

}

function listenForTransactions(callback) {
    console.log(`\$ {sender} starts reading transactions.`);
    const filtersByParty = {};
    filtersByParty[sender] = { inclusive: { templateIds: [PING, PONG] } }
};

const request = {
    begin: { boundary: da.LedgerOffset.Boundary.END },
    filter: { filtersByParty: filtersByParty }
};
const transactions = client.transactionClient.
getTransactions(request);
transactions.on('data', response => {
    for (const transaction of response.transactions) {
        const events = [];
        for (const event of transaction.events) {
            if (event.created) {
                events.push(event.created);
            }
        }
        if (events.length > 0) {
            callback(transaction.workflowId, events);
        }
    }
});
transactions.on('error', error => {
    console.error(`\$ {sender} encountered an error while processing
transactions!`);
    console.error(error);
    process.exit(-1);
});
}

function react(workflowId, events) {
    const reactions = [];
    for (const event of events) {
        const { receiver: { party: receiver }, count: { int64: count } } =
event.arguments.fields;
        if (receiver === sender) {
            const templateId = event.templateId;
            const contractId = event.contractId;
            const reaction = templateId.name == PING.name ? 'ReplyPong'
: 'ReplyPing';
            console.log(`\$ {sender} (workflow \$ {workflowId}): \$ {reaction} at count \$ {count}`);
            reactions.push({
                exercise: {
                    templateId: templateId,

```

(continues on next page)

(continued from previous page)

```
        contractId: contractId,
        choice: reaction,
        argument: { record: { fields: {} } }
    }
})
}
}

if (reactions.length > 0) {
    const request = {
        commands: [
            applicationId: 'PingPongApp',
            workflowId: workflowId,
            commandId: uuidv4(),
            ledgerEffectiveTime: { seconds: 0, nanoseconds: 0 },
            maximumRecordTime: { seconds: 5, nanoseconds: 0 },
            party: sender,
            list: reactions
        ]
    }
    client.commandClient.submitAndWait(request, error => {
        if (error) throw error;
    });
}
}
});
```

To test your code you need to run two different commands in two different terminals.

First, run:

```
npm start Alice Bob
```

After starting this, the application creates a ping contract on the ledger and waits for replies.

Alice starts reading transactions.  
Created Ping contract from Alice to Bob.

Keep this command running, open a new shell and run the following command:

```
npm start Bob Alice
```

You should now see the exchange happening on both terminals.

npm start Alice Bob

```
Alice starts reading transactions.  
Created Ping contract from Alice to Bob.  
Alice (workflow Ping-Bob): Pong at count 0  
Alice (workflow Ping-Bob): Pong at count 2
```

```
npm start Bob Alice
```

```
Bob starts reading transactions.
Created Ping contract from Bob to Alice.
Bob (workflow Ping-Bob): Ping at count 1
Bob (workflow Ping-Bob): Ping at count 3
```

You can now close both applications.

Your application is now able to complete the full exchange. Very well done!

## The Active Contracts Service

So far so good, but there is a flaw. You might have noticed that the application is subscribing for transactions using `boundary: da.LedgerOffset.Boundary.END`. This means that wherever the ledger is at that time, the application is going to see transactions only after that, missing contracts created earlier. This problem could be addressed by subscribing for transactions from `boundary: da.LedgerOffset.Boundary.BEGIN`, but then in case of a downtime your application would need to be prepared to handle contracts it has already processed before. To make this recovery easier the API offers a service which returns the set of active contracts on the ledger and an offset with which one can subscribe for transactions. This facilitates ramping up new applications and you can be sure to see contracts only once.

In this new example the application first processes the current active contracts. Since that process is asynchronous the rest of the program should be passed in as a callback.

```
function processActiveContracts(transactionFilter, callback, onComplete) {
    const request = { filter: transactionFilter };
    const activeContracts = client.activeContractsClient.
    ↪getActiveContracts(request);
    let offset = undefined;
    activeContracts.on('data', response => {
        if (response.activeContracts) {
            const events = [];
            for (const activeContract of response.activeContracts) {
                events.push(activeContract);
            }

            if (events.length > 0) {
                callback(response.workflowId, events);
            }
        }
    });

    if (response.offset) {
        offset = response.offset;
    }
});

activeContracts.on('error', error => {
    console.error(`\$ {sender} encountered an error while processing
    ↪active contracts!`);
    console.error(error);
```

(continues on next page)

(continued from previous page)

```

        process.exit(-1);
    });

    activeContracts.on('end', () => onComplete(offset));
}

```

Note that the transaction filter was factored out as it can be shared. The final code would look like this:

```

const da = require('@da/daml-ledger').data;
const templateIds = require('./template-ids.json');

const PING = templateIds['PingPong.Ping'];
const PONG = templateIds['PingPong.Pong'];

const uuidv4 = require('uuid/v4');

let [, , sender, receiver, host, port] = process.argv;
host = host || 'localhost';
port = port || 7600;
if (!sender || !receiver) {
    console.log('Missing sender and/or receiver arguments, exiting.');
    process.exit(-1);
}

da.DamlLedgerClient.connect({ host: host, port: port }, (error, client) =>
{
    if (error) throw error;

    const filtersByParty = {};
    filtersByParty[sender] = { inclusive: { templateIds: [PING, PONG] } };
    const transactionFilter = { filtersByParty: filtersByParty };

    processActiveContracts(transactionFilter, react, offset => {
        listenForTransactions(offset, transactionFilter, react);
        createFirstPing();
    });
}

function createFirstPing() {
    const request = {
        commands: {
            applicationId: 'PingPongApp',
            workflowId: `Ping-${sender}`,
            commandId: uuidv4(),
            ledgerEffectiveTime: { seconds: 0, nanoseconds: 0 },
            maximumRecordTime: { seconds: 5, nanoseconds: 0 },
            party: sender,
            list: [
            ]
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

        create: {
            templateId: PING,
            arguments: {
                fields: {
                    sender: { party: sender },
                    receiver: { party: receiver },
                    count: { int64: 0 }
                }
            }
        }
    ]
}
};

client.commandClient.submitAndWait(request, error => {
    if (error) throw error;
    console.log(`Created Ping contract from ${sender} to ${
    ↪{receiver}.`);
    });
}

function listenForTransactions(offset, transactionFilter, callback) {
    console.log(`${sender} starts reading transactions from offset: ${
    ↪{offset.absolute}.}`);
    const request = {
        begin: { boundary: da.LedgerOffset.Boundary.END },
        filter: transactionFilter
    };
    const transactions = client.transactionClient.
    ↪getTransactions(request);
    transactions.on('data', response => {
        for (const transaction of response.transactions) {
            const events = [];
            for (const event of transaction.events) {
                if (event.created) {
                    events.push(event.created);
                }
            }
            if (events.length > 0) {
                callback(transaction.workflowId, events);
            }
        }
    });
    transactions.on('error', error => {
        console.error(`${sender} encountered an error while processing
    ↪transactions!`);
        console.error(error);
        process.exit(-1);
    });
}

```

(continues on next page)

(continued from previous page)

```

    }

    function processActiveContracts(transactionFilter, callback,
→onComplete) {
    console.log(`processing active contracts for ${sender}`);
    const request = { filter: transactionFilter };
    const activeContracts = client.activeContractsClient.
→getActiveContracts(request);
    let offset = undefined;
    activeContracts.on('data', response => {
        if (response.activeContracts) {
            const events = [];
            for (const activeContract of response.activeContracts) {
                events.push(activeContract);
            }
            if (events.length > 0) {
                callback(response.workflowId, events);
            }
        }
        if (response.offset) {
            offset = response.offset;
        }
    });
}

activeContracts.on('error', error => {
    console.error(`${sender} encountered an error while processing
→active contracts!`);
    console.error(error);
    process.exit(-1);
});

activeContracts.on('end', () => onComplete(offset));
}

function react(workflowId, events) {
    const reactions = [];
    for (const event of events) {
        const { receiver: { party: receiver }, count: { int64: count } }
→} = event.arguments.fields;
    if (receiver === sender) {
        const templateId = event.templateId;
        const contractId = event.contractId;
        const reaction = templateId.name == PING.name ? 'ReplyPong'
→: 'ReplyPing';
        console.log(`${sender} (workflow ${workflowId}): $
→{reaction} at count ${count}`);
        reactions.push({
            exercise: {

```

(continues on next page)

(continued from previous page)

```
        templateId: templateId,
        contractId: contractId,
        choice: reaction,
        argument: { record: { fields: {} } }
    }
})
}
}

if (reactions.length > 0) {
  const request = {
    commands: {
      applicationId: 'PingPongApp',
      workflowId: workflowId,
      commandId: uuidv4(),
      ledgerEffectiveTime: { seconds: 0, nanoseconds: 0 },
      maximumRecordTime: { seconds: 5, nanoseconds: 0 },
      party: sender,
      list: reactions
    }
  };
  client.commandClient.submitAndWait(request, error => {
    if (error) throw error;
  });
}
}
});
```

Before running this you should start with a clean ledger to avoid being confused by the unprocessed contracts from previous examples.

da stop  
da sandbox

Then run:

npm start Alice Bob

in another shell:

```
npm start Bob Alice
```

You should see the following outputs respectively.

```
processing active contracts for Alice
Alice starts reading transactions from offset: 0.
Created Ping contract from Alice to Bob.
Alice (workflow Ping-Bob): Pong at count 0
Alice (workflow Ping-Alice): Ping at count 1
Alice (workflow Ping-Bob): Pong at count 2
Alice (workflow Ping-Alice): Ping at count 3
```

```

processing active contracts for Bob
Bob (workflow Ping-Alice): Pong at count 0
Bob starts reading transactions from offset: 1.
Created Ping contract from Bob to Alice.
Bob (workflow Ping-Bob): Ping at count 1
Bob (workflow Ping-Alice): Pong at count 2
Bob (workflow Ping-Bob): Ping at count 3

```

Alice joining an empty ledger has no active contracts to process. Bob however, who joins later, will see Alice's Ping contract and process it. Afterwards he will continue listening to transactions from offset 1.

### 7.1.1.3 DAML as JSON

The Node.js bindings accept as parameters and return as results plain JavaScript objects.

More specifically, these objects are subset that is fully compatible with the JSON data-interchange format. This means that any request and response to and from the ledger can be easily used with other services that accept JSON as an input format.

The [reference documentation](#) is more specific about the expected shape of objects sent to and received from the ledger, but to give you a sense of how these objects look, the following represents the creation of a `Pvp` record.

```

{
  create: {
    templateId: { packageId: '934023fa9c89e8f89b8a', name: 'Pvp.Pvp' },
    arguments: {
      recordId: { packageId: '934023fa9c89e8f89b8a', name: 'Pvp.Pvp'
    },
      fields: {
        buyer : { party: 'some-buyer' },
        seller : { party: 'some-seller' },
        baseIssuer : { party: 'some-base-issuer' },
        baseCurrency : { text: 'CHF' },
        baseAmount : { decimal: '1000000.00' },
        baseIouCid : { variant: { variantId: { packageId:
          ↵'ba777d8d7c88e87f7', name: 'Maybe' }, constructor: 'Just', value: {
          ↵contractId: '76238b8998a98d98e978f' } } },
          quoteIssuer : { party: 'some-quote-issuer' },
          quoteCurrency: { text: 'USD' },
          quoteAmount : { decimal: '1000001.00' },
          quoteIouCid : { variant: { variantId: { packageId:
            ↵'ba777d8d7c88e87f7', name: 'Maybe' }, constructor: 'Just', value: {
              ↵contractId: '76238b8998a98d98e978f' } } },
              settleTime : { timestamp: 93641099000000000
            }
          }
        }
      }
}

```

Notice that fields of a template are represented as keys in fields. Each value is then another object, where the key is the type. This is necessary for the ledger to disambiguate between, for example, strings that represent text and strings that represent a decimal.

The Node.js bindings let you access the Ledger API from Node.js code.

The Installing page guides you through the process of [Getting started](#).

After that, a tutorial teaches you [how to build a simple application from scratch](#).

Finally, [the API reference documentation](#) and [an introduction to DAML as JSON](#) are also available.

## 7.1.2 Navigator Console

The Navigator Console is a terminal-based front-end for inspecting and modifying a Digital Asset ledger. It's useful for DAML developers, app developers, or business analysts who want to debug or analyse a ledger by exploring it manually.

You can use the Console to:

- inspect available templates
- query active contracts
- exercise commands
- list blocks and transactions

If you prefer to use a graphical user interface for these tasks, use the [Navigator](#) instead.

### On this page:

[Try out the Navigator Console on the Quickstart](#)

- [Installing and starting Navigator Console](#)
- [Getting help](#)
- [Exiting Navigator Console](#)
- [Using commands](#)

[Displaying status information](#)

[Choosing a party](#)

[Advancing time](#)

[Inspecting templates](#)

[Inspecting contracts, transactions, and events](#)

[Querying data](#)

[Creating contracts](#)

[Exercising choices](#)

- [Advanced usage](#)

[Using Navigator outside the SDK](#)

[Using Navigator with the Digital Asset ledger](#)

### 7.1.2.1 Try out the Navigator Console on the Quickstart

With the sandbox running the [quickstart application](#)

1. To start the shell, run `da run navigator -- console localhost 7600`

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 [ou] template:

get the full name of the location.

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@317057d06d4bc4bb91bf3cfe3292bf3c2467c5e004290e0ba20b  
with {issuer="Bob", owner="Bob", currency="BobsCoin", amount="1.0",  
observers=[]}
```

You should see the following output:

```
CommandId: 1b8af77a91ad1102  
Status: Success  
TransactionId: 10
```

4. You can see details of that contract using the TransactionId. First, run:

transaction 10

to get details of the transaction that created the contract:

```
Offset: 11
Effective at: 1970-01-01T00:00:00Z
Command ID: 1b8af77a91ad1102
Events:
- [#10:0] Created #10:0 as IoU
```

Then, run:

contract #10:0

to see the contract for the new BobsCoin:

Id: #10:0

TemplateId:

→ IoU@317057d06d4bc4bb91bf3cfe3292bf3c2467c5e004290e0ba20b993eb1e40931

#### Argument:

observers:

issuer: Bob

amount: 1.0

currency: BobsCoin

owner: Bob

Created:

EventId: #10·0

(continues on next page)

(continued from previous page)

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 `lou` contracts by running `sql select * from contract where template_id like 'Iou.Iou@%'`. For more examples, take a look at the [Navigator Console database documentation](#).

For a full list of commands, run `help`. You can also look at the [Navigator Console documentation page](#).

For help on a particular command, run `help <name of command>`.

When you are done exploring the shell, run quit to exit.

## Installing and starting Navigator Console

Navigator Console is installed as part of the DAML SDK. See [Installing the SDK](#) for instructions on how to install the DAML SDK.

If you want to use Navigator Console independent of the SDK, see the [Advanced usage](#) section.

To run Navigator Console:

1. Open a terminal window and navigate to your DAML SDK project folder.
  2. If the Sandbox isn't already running, run it with the command `da start`.  
The sandbox prints out the port on which it is running - by default, port 6865.
  3. Run `da run navigator -- console localhost 6865`. Replace 6865 by the port reported by the sandbox, if necessary.

When Navigator Console starts, it displays a welcome message:

Version X.Y.Z

Welcome to the console. Type 'help' to see a list of commands.

## Getting help

To see all available Navigator Console commands and how to use them, use the `help` command:

```
>help
Available commands:
choice          Print choice details
```

(continues on next page)

(continued from previous page)

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 <b>all</b> DAML-LF packages
parties	List <b>all</b> parties available <b>in</b> Navigator
party	Set the current party
quit	Quit the application
set_time	Set the (static) ledger effective time
templates	List <b>all</b> 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
```

## Exiting Navigator Console

To exit Navigator Console, use the `quit` command:

```
>quit
Bye.
```

## Using commands

This section describes how to use some common commands.

**Note:** Navigator Console has several features to help with typing commands:

Press the **Tab** key one or more times to use auto-complete and see a list of suggested text to complete the command.

Press the **Up** or **Down** key to scroll through the history of recently used commands.

Press **Ctrl+R** to search in the history of recently used commands.

### 7.1.2.2 Displaying status information

To see useful information about the status of both Navigator Console and the ledger, use the `info` command:

(continues on next page)

(continued from previous page)

```
Active contracts: 1001
Last transaction: scenario-transaction-2002
```

### 7.1.2.3 Choosing a party

Privacy is an important aspect of a Digital Asset ledger: parties can only access the contracts on the ledger that they are authorized to. This means that, before you can interact with the ledger, you must assume the role of a particular party.

The currently active party is displayed left of the prompt sign (>). To assume the role of a different party, use the `party` command:

```
BANK1>party BANK2
BANK2>
```

---

**Note:** The list of available parties is configured when the Sandbox starts. (See the [SDK Assistant](#) or [Advanced usage](#) for more instructions.)

---

### 7.1.2.4 Advancing time

You can advance the time of the DAML Sandbox. This can be useful when testing, for example, when entering a trade on one date and settling it on a later date.

(For obvious reasons, this feature does not exist on the Digital Asset ledger.)

To display the current ledger time, use the `time` command:

```
>time
1970-01-01T00:16:40Z
```

To advance the time to the time you specify, use the `set_time` command:

```
>set_time 1970-01-02T00:16:40Z
New ledger effective time: 1970-01-02T00:16:40Z
```

### 7.1.2.5 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
```

### 7.1.2.6 Inspecting contracts, transactions, and events

The ledger is a record of transactions between authorized participants on the distributed network. Transactions consist of events that create or archive contracts, or exercise choices on them.

To get detailed information about a ledger object, use the singular form of the command (`transaction`, `event`, `contract`):

```
>transaction 2003
Offset: 1004
Effective at: 1970-01-01T00:16:40Z
Command ID: 732f6ac4a63c9802
Events:
- [#2003:0] Created #2003:0 as RightOfUseOffer
```

```
>event #2003:0
Id: #2003:0
ParentId: ???
TransactionId: 2003
WorkflowId: e13067beec13cf4c
Witnesses:
- Scrooge_McDuck
Type: Created
Contract: #2003:0
Template: Main.RightOfUseOffer
```

(continues on next page)

(continued from previous page)

**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:
```

### 7.1.2.7 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
```

<code>id</code>	<code>template_id</code>	<code>archive_transaction_id</code>
#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.

---

### 7.1.2.8 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 using the same syntax as in the DAML language:

```
>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:  
  Status: Success  
  TransactionId: 2005
```

### 7.1.2.9 Exercising choices

To exercise a choice on a contract with the given ID, use the `exercise` command:

```
>exercise #2005:0 Accept  
CommandID: 8dbbc917c7beee  
Status: Success  
TransactionId: 2006
```

```
>exercise #2005:0 Accept with {tenant="BANK2"}  
CommandID: 8dbbc917c7beee
```

(continues on next page)

(continued from previous page)

Status: Success
TransactionId: 2006

## Advanced usage

### 7.1.2.10 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> } .. }
--

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" } }
---

### 7.1.2.11 Using Navigator with the Digital Asset ledger

By default, Navigator is configured to use an unencrypted connection to the ledger.

To run Navigator against a secured Digital Asset Ledger, configure TLS certificates using the `--pem`, `--crt`, and `--cacrt` command line parameters.

Details of these parameters are explained in the command line help:

```
da run navigator -- --help
```

## 7.1.3 Querying the Navigator local database

You can query contracts, transactions, events, or commands in any way you'd like, by querying the Navigator Console's local database(s) directly. This page explains how you can run queries.

---

**Note:** Because of the strong DAML privacy model, each party will see a different subset of the ledger data. For this reason, each party has its own local database.

---

The Navigator database is implemented on top of [SQLite](#). SQLite understands most of the standard SQL language. For information on how to compose SELECT statements, see to the SQLite [SELECT syntax specification](#).

To run queries, use the `sql` Navigator Console command. Take a look at the examples below to see how you might use this command.

#### On this page:

- [How the data is structured](#)
- [Example query using plain SQL](#)
- [Example queries using JSON functions](#)

### 7.1.3.1 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](#).

### 7.1.3.2 Example query using plain SQL

Filter on the template id of contracts:

```
sql select count (*) from contract where template_id like '%Offer%'
```

### 7.1.3.3 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'
```

## 7.1.4 Extractor

### 7.1.4.1 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)

#### 7.1.4.2 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:

```
$ da run extractor -- --help
```

#### 7.1.4.3 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 7600  
to PostgreSQL instance running on localhost  
identified by the user postgres without a password set  
into a database called daml\_export

```
$ da run extractor -- postgresql --user postgres --connecturl  
→ jdbc:postgresql:daml_export --party Scrooge_McDuck -h 192.168.1.12 -p  
→ 7600 --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.

#### 7.1.4.4 Running the Extractor

The basic command to run the Extractor is:

```
$ da run extractor -- [options]
```

For what options to use, see the next sections.

### 7.1.4.5 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 7600`, or in short form `-h 10.1.1.168 -p 7600`.

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.

### 7.1.4.6 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

```
$ da run 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`

```
$ da run extractor -- postgres --connecturl jdbc:postgresql://192.168.1.
 ↪12:5432/daml_export --user daml_exporter --password ExamplePassword --
 ↪party [party]
```

### 7.1.4.7 Full list of options

To see the full list of options, run the `--help` command, which gives the following output:

```
Usage: extractor [prettyprint|postgresql] [options]

Command: prettyprint [options]
Pretty print contract template and transaction data to stdout.
--width <value>           How wide to allow a pretty-printed value to
↪become before wrapping.
                           Optional, default is 120.
--height <value>          How tall to allow each pretty-printed output to
↪become before
                           it is truncated with a `...`.
                           Optional, default is 1000.
```

(continues on next page)

(continued from previous page)

```

Command: postgresql [options]
Extract data into a PostgreSQL database.
  --connecturl <value>      Connection url for the `org.postgresql.Driver`
  ↵driver. For examples,
                                visit https://jdbc.postgresql.org/documentation/
  ↵80/connect.html
  --user <value>            The database user on whose behalf the connection
  ↵is being made.
  --password <value>        The user's password. Optional.

Common options:
  -h, --ledger-host <h>    The address of the Ledger host. Default is 127.
  ↵0.0.1
  -p, --ledger-port <p>    The port of the Ledger host. Default is 6865.
  --party <value>          The party whose contract data should be
  ↵extracted.
  --from <value>           The transaction offset (exclusive) for the
  ↵snapshot start position.
                                Must not be greater than the current latest
  ↵transaction offset.
                                Optional, defaults to the beginning of the
  ↵ledger.
                                Currently, only the integer-based Sandbox
  ↵offsets are supported.
  --to <value>              The transaction offset (inclusive) for the
  ↵snapshot end position.
                                Use "head" to use the latest transaction offset
  ↵at the time
                                the extraction first started, or "follow" to
  ↵stream indefinitely.
                                Must not be greater than the current latest
  ↵offset.
                                Optional, defaults to "follow".
  --help                    Prints this usage text.

TLS configuration:
  --pem <value>             TLS: The pem file to be used as the private key.
  --crt <value>              TLS: The crt file to be used as the cert chain.
  Required if any other TLS parameters are set.
  --cacrt <value>            TLS: The crt file to be used as the the trusted
  ↵root CA.

```

Some options are tied to a specific subcommand, like `--connecturl` only makes sense for the `postgresql`, while others are general, like `--party`.

#### 7.1.4.8 Output format

To understand the format that Extractor outputs into a PostgreSQL database, you need to understand how the ledger stores data.

The DAML Ledger is composed of transactions, which contain events. Events can represent:

- creation of contracts (create event), or
- exercise of a choice on a contract (exercise event).

A contract on the ledger is either active (created, but not yet archived), or archived. The relationships between transactions and contracts are captured in the database: all contracts have pointers (foreign keys) to the transaction in which they were created, and archived contracts have pointers to the transaction in which they were archived.

#### 7.1.4.9 Transactions

Transactions are stored in the `transaction` table in the `public` schema, with the following structure

```
CREATE TABLE transaction
  (transaction_id TEXT PRIMARY KEY NOT NULL
  ,seq BIGSERIAL UNIQUE NOT NULL
  ,workflow_id TEXT
  ,effective_at TIMESTAMP NOT NULL
  ,extracted_at TIMESTAMP DEFAULT NOW()
  ,ledger_offset TEXT NOT NULL
);
```

**transaction\_id**: The transaction ID, as appears on the ledger. This is the primary key of the table.

**transaction\_id, effective\_at, workflow\_id, ledger\_offset**: These columns are the properties of the transaction on the ledger. For more information, see the [specification](#).

**seq**: Transaction IDs should be treated as arbitrary text values: you can't rely on them for ordering transactions in the database. However, transactions appear on the Ledger API transaction stream in the same order as they were accepted on the ledger. You can use this to work around the arbitrary nature of the transaction IDs, which is the purpose of the `seq` field: it gives you a total ordering of the transactions, as they happened from the perspective of the ledger. Be aware that `seq` is not the exact index of the given transaction on the ledger. Due to the privacy model of the DAML Ledger, the transaction stream won't deliver a transaction which doesn't concern the party which is subscribed. The transaction with `seq` of 100 might be the 1000th transaction on the ledger; in the other 900, the transactions contained only events which mustn't be seen by you.

**extracted\_at**: The `extracted_at` field means the date the transaction row and its events were inserted into the database. When extracting historical data, this field will point to a possibly much later time than `effective_at`.

#### 7.1.4.10 Contracts

Create events and contracts that are created in those events are stored in the `contract` table in the `public` schema, with the following structure

```
CREATE TABLE contract
  (event_id TEXT PRIMARY KEY NOT NULL
  ,archived_by_event_id TEXT DEFAULT NULL
  ,contract_id TEXT NOT NULL
```

(continues on next page)

(continued from previous page)

```
,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](#).

#### 7.1.4.11 Exercises

Exercise events are stored in the `exercise` table in the `public` schema, with the following structure:

```
CREATE TABLE
exercise
(event_id TEXT PRIMARY KEY NOT NULL
,transaction_id TEXT NOT NULL
,is_root_event BOOLEAN NOT NULL
,contract_id TEXT NOT NULL
,package_id TEXT NOT NULL
,template TEXT NOT NULL
,contract_creating_event_id TEXT NOT NULL
,choice TEXT NOT NULL
,choice_argument JSONB NOT NULL
,acting_parties JSONB NOT NULL
,consuming BOOLEAN NOT NULL
,witness_parties JSONB NOT NULL
,child_event_ids JSONB NOT NULL
);
```

**package\_id, template:** The fields `package_id` and `template` are the exploded version of the `template_id` property of the ledger event.

**is\_root\_event:** Indicates whether the event in which the contract was created was a root event of the corresponding transaction.

**transaction\_id:** The `transaction_id` field refers to the transaction in which the contract was created.

The other columns are properties of the `ExercisedEvent` class in a transaction. For more information, see the [specification](#).

#### 7.1.4.12 JSON format

Values on the ledger can be either primitive types, user-defined records, or variants. An extracted contract is represented in the database as a record of its create argument, and the fields of that record are either primitive types, other records, or variants. A contract can be a recursive structure of arbitrary depth.

These types are translated to [JSON types](#) the following way:

##### Primitive types

`ContractID`: represented as [string](#).

`Int64`: represented as [string](#).

`Decimal`: A decimal value with precision 38 (38 decimal digits), of which 10 after the comma / period. Represented as [string](#).

`List`: represented as [array](#).

`Text`: represented as [string](#).

`Date`: days since the unix epoch. represented as [integer](#).

`Time`: Microseconds since the UNIX epoch. Represented as [number](#).

`Bool`: represented as [boolean](#).

`Party`: represented as [string](#).

`Unit` and `Empty` are represented as empty records.

`Optional`: represented as [object](#), as it was a Variant with two possible constructors: `None` and `Some`.

##### User-defined types

`Record`: represented as [object](#), where each create parameter's name is a key, and the parameter's value is the JSON-encoded value.

`Variant`: represented as [object](#), using the `{constructor: body}` format, e.g. `{"Left": true}`.

#### 7.1.4.13 Examples of output

The following examples show you what output you should expect.

The examples use the default built-in SDK project which you get by running `$ da new [project_name]`. The Sandbox has already run the scenarios of the DAML model which 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:

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.

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

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", "addr...}	[{"McDuck Man", {"Betina_Beakley"}]
#scenario-transaction-1:1:1	NULL	#1:1	scenario-transaction-1	NULL	FALSE	528d2184c218aa9b0b960db7882cd5abc6d ba83078aab1dc8e9dbe6860a5a548	Main:RightOfUseAgreement	{"tenant": "Betina_Beakley", "addr...}	[{"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	528d2184c218aa9b0b960db7882cd5abc6d 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.

#### 7.1.4.14 Dealing with schema evolution

When updating packages, you can end up with multiple versions of the same package in the system.

Let's say you have a template called My.Company.Finance.Account:

```
daml 1.2 module My.Company.Finance.Account where

template Account
  with
    provider: Party
    accountId: Text
    owner: Party
    observers: [Party]
  where
    [...]
```

This is built into a package with a resulting hash 6021727fe0822d688ddd545997476d530023b222d02f191

Later you add a new field, displayName:

```
daml 1.2 module My.Company.Finance.Account where

template Account
  with
    provider: Party
    accountId: Text
    owner: Party
    displayName: Text
```

(continues on next page)

(continued from previous page)

```
observers: [Party]
displayName: Text
where
[...]
```

The hash of the new package with the update is 1239d1c5df140425f01a5112325d2e4edf2b7ace223f8c1d

There are contract instances of first version of the template which were created before the new field is added, and there are contract instances of the new version which were created since. Let's say you have one instance of each:

```
{
  "owner": "Bob",
  "provider": "Bob",
  "accountId": "6021-5678",
  "observers": [
    "Alice"
  ]
}
```

and:

```
{
  "owner": "Bob",
  "provider": "Bob",
  "accountId": "1239-4321",
  "observers": [
    "Alice"
  ],
  "displayName": "Personal"
}
```

They will look like this when extracted:

<b>id</b>	<b>seq</b>	<b>event_id</b>	<b>transaction_id</b>	<b>archived_by_transaction_id</b>	<b>package_id</b>	<b>template</b>	<b>contract</b>
#3:0	3	#3:0	3	NULL	1239d1c5df140425f01a5112325d2e4ed f2b7ace223fb8c1d2ebebe7ba8ececfe	My.Company.Finance.Account	{"owner": "Bob", "provider": "Bob", "accountId": "1239-4321", "observers": ["Alice"], "displayName": "Personal"}
#4:0	4	#4:0	4	NULL	6021727fe0822d688dd545997476d53 0023b22d02f1919567bd82b205a5ce3	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
```

(continues on next page)

(continued from previous page)

```

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"]

### 7.1.4.15 Logging

By default, the Extractor logs to stderr, with INFO verbose level. To change the level, use the `-DLOGLEVEL=[level]` option, e.g. `-DLOGLEVEL=TRACE`.

You can supply your own logback configuration file via the standard method: <https://logback.qos.ch/manual/configuration.html>

### 7.1.4.16 Continuity

When you terminate the Extractor and restart it, it will continue from where it left off. This happens because, when running, it saves its state into the state table in the public schema of the database. When started, it reads the contents of this table. If there's a saved state from a previous run, it restarts from where it left off. There's no need to explicitly specify anything, this is done automatically.

DO NOT modify content of the state table. Doing so can result in the Extractor not being able to continue running against the database. If that happens, you must delete all data from the database and start again.

If you try to restart the Extractor against the same database but with different configuration, you will get an error message indicating which parameter is incompatible with the already exported data. This happens when the settings are incompatible: for example, if previously contract data for the party Alice was extracted, and now you want to extract for the party Bob.

The only parameters that you can change between two sessions running against the same database are the connection parameters to both the ledger and the database. Both could have moved to different addresses, and the fact that it's still the same Ledger will be validated by using the Ledger ID (which is saved when the Extractor started its work the first time).

#### 7.1.4.17 Fault tolerance

Once the Extractor connects to the Ledger Node and the database and creates the table structure from the fetched DAML packages, it wraps the transaction stream in a restart logic with an exponential backoff. This results in the Extractor not terminating even when the transaction stream is aborted for some reason (the ledger node is down, there's a network partition, etc.).

Once the connection is back, it continues the stream from where it left off. If it can't reach the node on the host/port pair the Extractor was started with, you need to manually stop it and restart with the updated address.

Transactions on the ledger are inserted into PostgreSQL as atomic SQL transactions. This means either the whole transaction is inserted or nothing, so you can't end up with inconsistent data in the database.

#### 7.1.4.18 Troubleshooting

##### Can't connect to the Ledger Node

If the Extractor can't connect to the Ledger node on startup, you'll see a message like this in the logs, and the Extractor will terminate:

```
16:47:51.208 ERROR c.d.e.Main$@[kka.actor.default-dispatcher-7] - FAILURE:  
io.grpc.StatusRuntimeException: UNAVAILABLE: io exception.  
Exiting...
```

To fix this, make sure the Ledger node is available from where you're running the Extractor.

##### Can't connect to the database

If the database isn't available before the transaction stream is started, the Extractor will terminate, and you'll see the error from the JDBC driver in the logs:

```
17:19:12.071 ERROR c.d.e.Main$@[kka.actor.default-dispatcher-5] - FAILURE:  
org.postgresql.util.PSQLException: FATAL: database "192.153.1.23:daml_  
→export" does not exist.  
Exiting...
```

To fix this, make sure make sure the database exists and is available from where you're running the Extractor, the username and password your using are correct, and you have the credentials to connect to the database from the network address where the you're running the Extractor.

If the database connection is broken while the transaction stream was already running, you'll see a similar message in the logs, but in this case it will be repeated: as explained in the [Fault tolerance](#) section, the transaction stream will be restarted with an exponential backoff, giving the database,

network or any other trouble resource to get back into shape. Once everything's back in order, the stream will continue without any need for manual intervention.

## Chapter 8

# Support and Updates

## 8.1 Support and feedback

Have questions or feedback? You're in the right place.

**Questions: Stack Overflow** For how do I?, why does something work this way or I've got a programming problem I'm trying to solve questions, the [daml tag on Stack Overflow](#) is the best place to ask.

If you're not sure what makes a good question, take a look at [this checklist](#).

**Help and feedback: Slack, or Support widget** If you want to give feedback, or have questions that aren't right for Stack Overflow, you can:

Join the DAML community on [Slack](#).

Ask us for help using the [Support](#) link on any documentation page.

**Feedback: Email** If you'd rather use email, you can contact us at [sdk-feedback@digitalasset.com](mailto:sdk-feedback@digitalasset.com).

## 8.2 Release notes

This page contains release notes for the SDK.

DAML standard library (breaking change): Removed `DA.List.split` function, which was never intended to be exposed and doesn't do what the name suggests.

Java Bindings (breaking change): Removed type parameter for `DamlList` and `DamlOptional` classes. The `DamlList`, `DamlOptional`, and `ContractId` classes were previously parameterized (i.e `DamlList[String]`) for consistency with the DAML language. The type parameter has been removed as such type information is not supported by the underlying Ledger API and therefore the parameterized type couldn't be checked for correctness.

Java Bindings (breaking change): For all classes in the package `com.daml.ledger.javaapi.data`, we shortened the names of the conversion methods from long forms like `fromProtoGeneratedCompletionStreamRequest` and `toProtoGeneratedCompletionStreamRequest` to the much shorter `fromProto` and `toProto`.

Navigator: Add support for Optional and recursive data types.

Navigator: Improve start up performance for big DAML models.

Navigator (breaking change): Refactor the GraphQL API. If you're maintaining a modified version of the Navigator frontend, you'll need to adapt all your GraphQL queries to the new API.

DAML syntax (breaking change) : For the time being, datatypes with a single data constructor not associated with an argument are not accepted. For example `data T = T`. To workaround this use `data T = T {}` or `data T = T ()` (depending on whether you desire `T` be interpreted as a product or a sum).

### 8.2.1 0.11.3

Released on 2019-02-07

Changes:

Navigator: Fix display of Date values.

Extractor: Add first version of Extractor with PostgreSQL support.

### 8.2.2 0.11.2

Released on 2019-01-31

Changes:

Navigator: Add a terminal-based console interface using SQLite as a backend.

Navigator: Now writes logs to `./navigator.log` by default using Logback.

DAML Studio: Significant performance improvements.

DAML Studio: New table view for scenario results.

DAML Standard Library: New type classes.

Node.js bindings: Documentation updated to use version 0.4.0 and DAML 1.2.

### 8.2.3 0.11.1

Released on 2019-01-24

Changes:

Java Bindings: Fixed `Timestamp.fromInstant` and `Timestamp.toInstant`.

Java Bindings: Added `Timestamp.getMicroseconds`.

### 8.2.4 0.11.0

Released on 2019-01-17

Changes:

Documentation: [DAML documentation](#) and [examples](#) now use DAML 1.2.

To convert your code to DAML 1.2, see the [conversion guide](#).

Documentation: Added a comprehensive [quickstart guide](#) that replaces the old My first project example.

As part of this, removed the My first project, IOU and PvP examples.

Documentation: Added a [guide to building applications against a DA ledger](#).

Documentation: Updated the [support and feedback page](#).

Ledger API: Version 1.4.0 has support for multi-party subscriptions in the transactions and active contracts services.

Ledger API: Version 1.4.0 supports the verbose field in the transactions and active contracts services.

Ledger API: Version 1.4.0 has full support for transaction trees.

Sandbox: Implements Ledger API version 1.4.0.

Java Bindings: Examples updated to use version 2.5.2 which implements Ledger API version 1.4.0.

## 8.3 DAML SDK roadmap (as of April 2019)

This page specifies the major features we're planning to add next to the DAML SDK. Plans and timelines are subject to change. If you need any of these features or want to request others, see the [Support and feedback](#) page for how to get in touch.

We plan to update this roadmap roughly every three months.

### **Windows support**

Support for installing and running the SDK on Windows.

[More about this on GitHub](#).

### **Java ecosystem**

Improve the [Java bindings](#) and add code generation that generates Java classes from DAML types.

[More about this on GitHub](#).

### **JavaScript / TypeScript ecosystem**

Improve the currently experimental [JavaScript bindings](#) so they are stable, and add TypeScript code generation to generate code from DAML types.

[More about this on GitHub](#).

### **Simplified da assistant**

Rewritten [command line for the SDK](#) with improved usability.

[More about this on GitHub](#).

### **Native installers**

Allow users to install the SDK using native installers like homebrew and apt-get.

[More about this on GitHub](#).

### **Ledger SQL backend**

Replace the in-memory store used by the Sandbox with a SQL backend, so it's not just a development tool but also a persistent ledger you could deploy.

[More about this on GitHub](#).

### **Contract keys in SDK**

Contract keys are a subset of fields in a contract that allow you to look it up uniquely. Building on top of the experimental contract keys feature in DAML, give contract keys full SDK support and write documentation.

[More about this on GitHub](#).

### **Map and Enum types in DAML-LF**

Add Map and Enum types to DAML-LF (which is what DAML gets compiled to - it's used by the Ledger API).

[More about Map on GitHub](#).

[More about Enum on GitHub](#).

### **Better package management**

Make it easier to create packages and use packages.

[More about this on GitHub](#).

### **Web IDE**

Provide a browser-based version of [DAML Studio](#) to make it easier to try DAML out.

[More about this on GitHub](#).

### **DAML-on-X self-service package**

Make it easier for external developers to integrate DAML with other ledgers.

[More about this on GitHub.](#)