Project Structure

Canton Network Quickstart Guide | 2025

Version: 1.1.0-2025-02-17

Contents

Project Structure Overview

This document is intended to support **onboarding of an engineering team to Canton Network** with limited knowledge of Canton and Daml technologies. Canton Network Quickstart is not a **"platform"** or a **"product"**, instead it is **"bootstrap scaffolding"** providing an example build toolchain as well as This guide has a "full-stack developer view" to jump start your development.

As a guide to Quickstart, the treatment of supporting topics is necessarily shallow. In addition to the online documentation[1], Digital Asset provides a number of free courses[2] covering the Daml Language, the Canton Ledger, and architectural considerations. In particular the Technical Solution Architect path[3] provides a lot of useful background to this guide.

The Canton Quickstart Example Application

Business Case

The Canton Quickstart contains an example application providing both a demonstration of a Canton application targeted at production; and, a way of exercising the supporting developer scaffolding provided by the bootstrap.

This example application is a simple license management application that allows the application provider to issue licenses to application users. Canton Coin is used in this transaction. These users are assumed to be potentially retail customers of the provider, with access to Canton and a Canton Wallet, but not necessarily running their own infrastructure beyond a (possibly outsourced) validator node.

The relevant business entities are:

Amulet: An infrastructure token usable on the Canton synchronizer being used by this application. In the case of an application using Canton Network, this will be Canton Coin[4].

DSO Party: The Decentralized Synchronizer Operations Canton Party. This is the party that operates the Amulet token in which the provider accepts license payments. In the case of a Canton Network Application, this will be the Global Synchronizer Foundation.

Application Provider: This is a Canton Party representing the legal entity deploying, running, and offering the application to their users (customers). In a licensing application, this is the entity offering to sell the licenses.

[1] https://docs.daml.com/, and https://dev.network.canton.global/index.html
[2] https://www.digitalasset.com/training-and-certification
[3] https://daml.talentlms.com/catalog/info/id:160
[4] https://www.canton.network/blog/canton-coin-a-canton-network-native-payment-application

Application User: This is a Canton Party representing the legal entity that is (presumably) a customer of the application provider. In this application this is an entity with a need to purchase a license, and periodically renew it. Canton Coin is exchanged for the license.

Core Workflows Happy Path Business Requirement

Issuing a License

Given an application user (app-user) has been onboarded to the licensing application

When the application provider (app-provider) instructs the application to create a new license for the app-user

Then a new expired license will be created on the ledger and made visible to the app-user

Requesting a License Renewal

Given an app-user has a license (I1)

And the current datetime is greater than the expiration date on license (I1)

When the app-provider instructs the application to request a license renewal

Then a license renewal will be created and made visible to the user

And a matching amulet (canton coin) payment request will be created on the ledger

Paying for a License Renewal

Given there is a license renewal request on the ledger

And a matching amulet payment request on the ledger

When the user indicates they wish to pay the renewal

Then the user will be redirected to the payment request in their wallet, and can approve the request

Renewing the license

Given an app-user has approved an amulet payment request associated with a license renewal request

And there is an AcceptedAppPayment contract (accepted-payment) on the ledger corresponding to that approval

When the app-provider instructs the application to complete the renewal transaction

Then the license will be updated with a new expiration date = renewal duration + max (old expiration date, now)

And the app-provider will exercise the AcceptedAppPayment_Collect choice on accepted-payment

Topology

The Canton Quickstart project bootstrap provides two "deployment" modes: DevNet and LocalNet. The local topology of the DevNet configuration is intended for testing against the Canton DevNet, and provides only those nodes that would be necessary to run the application in production. The LocalNet configuration is a superset of DevNet and also runs local versions of a super validator, and the canton coin wallet application. This allows running/testing/demonstrating the application entirely on a single machine. Please note that this version of LocalNet will need considerable memory resources[5].

In addition to the DevNet vs LocalNet distinction, there is also an optional Observability configuration that will work with either mode. This will bring up a fully configured OpenTelemetry deployment with metrics, monitoring, log aggregation, and trace analysis.

Running `make setup` in the `quickstart/` directory will allow you to select between these various configurations.

Once your configuration is built and running, `make status` in `quickstart/` will display the associated running docker containers.

See the Topology documentation in the `docs/` directory for more detailed information on the various nodes in each configuration and their relationship to each other/.

Top level

Most of the top level project directory is associated with supporting a portable, consistent cross-platform development environment. It does this using the package manager Nix[6], Direnv[7], and Docker Compose[8]. The toplevel setup ensures a consistent and repeatable dev, build, and test regardless of choice of environment.

[5] While writing this guide, the author's Docker configuration was 10 CPUs & 25GB RAM
[6] https://nixos.org/download/
[7] https://direnv.net/
[8] https://docs.docker.com/compose/

The current toplevel directory contents for a fresh checkout include:

```
√ % ls -lAgo
total 32
-rw-r--r--  1  427 Feb 11 17:20 .envrc
drwxr-xr-x 12  384 Feb 11 17:23 .git
-rw-r--r--  1  214 Feb 11 17:20 .gitattributes
drwxr-xr-x  3   96 Feb 11 17:20 .github
-rw-r--r--  1  587 Feb 11 17:20 .gitignore
-rw-r--r--  1  680 Feb 11 17:20 LICENSE
-rw-r--r--  1 6592 Feb 11 17:20 README.md
-rw-r--r--  1  702 Feb 11 17:20 SECURITY.md
drwxr-xr-x  4  128 Feb 11 17:20 docs
drwxr-xr-x  4  128 Feb 11 17:20 nix
drwxr-xr-x 18  576 Feb 11 17:20 quickstart
-rw-r--r--  1  881 Feb 11 17:20 shell.nix
```

`.git*` The usual git files and directories. In particular, `.gitignore` is configured to exclude build artifacts for the current build systems in use; Daml SDK support files; and, IDE project artifacts for Visual Code or other IDEs.

`.envrc` This is a part of the Direnv configuration. Specifically it activates the Nix environment for the project via a call to `use nix` which uses the `shell.nix` file to set up the development environment using nix-shell[9].

`LICENSE`, `Security.md`, and `README.md`. The License is 0BSD.

`docs/` contains some engineering documentation for the example app.

`quickstart/` is the main project directory. If you do not wish to use Nix, this directory can be made the toplevel directory for your project — although you will then need to manage your binary dependencies manually. The next section covers this directory in detail.

`shell.nix`[10] and `nix/` contain the Nix configuration. Familiarity with shell.nix is essential, as it manages new dependencies. Note `nix/sources.json` pins the nix release to ensure determinacy across builds. You will want to ensure this gets updated at an appropriate cadence that balances staying up to date with development environment stability.

[9] https://nixos.wiki/wiki/Development_environment_with_nix-shell
[10] https://nix.dev/tutorials/first-steps/declarative-shell.html

Current Dependencies declared in `shell.nix`

- npins
- jdk17
- nodejs_18
- typescript

These are in addition to the Nix stdenv environment[11].

Quickstart Project Directory

As is typical in a project directory the files and directories here fall into one of three categories:

- Build configuration
- Deployment configuration
- Application source

```
√ % ls -lAgo
total 124
-rw-r--r--  1  5665 Feb 11 17:20 .env
-rw-r--r--  1  7689 Feb 11 17:20 Makefile
-rw-r--r--  1 58076 Feb 11 17:20 NOTICES
drwxr-xr-x  5   160 Feb 11 17:20 backend
-rw-r--r--  1   918 Feb 11 17:20 build.gradle.kts
drwxr-xr-x  5   160 Feb 11 17:20 buildSrc
drwxr-xr-x  3    96 Feb 11 17:20 common
-rw-r--r--  1 21611 Feb 11 17:20 compose.yaml
drwxr-xr-x  5   160 Feb 11 17:20 config
drwxr-xr-x 10   320 Feb 11 17:20 daml
drwxr-xr-x 16   512 Feb 11 17:20 docker
drwxr-xr-x 14   448 Feb 11 17:20 frontend
drwxr-xr-x  4   128 Feb 11 17:20 gradle
-rwxr-xr-x  1  8706 Feb 11 17:20 gradlew
-rw-r--r--  1  2918 Feb 11 17:20 gradlew.bat
-rw-r--r--  1   670 Feb 11 17:20 settings.gradle.kts
```

[11] https://nixos.org/manual/nixpkgs/stable/#sec-tools-of-stdenv

Build Configuration

The primary build tool used by the example project is Gradle. As is recommended, this is managed via the Gradle wrappers `gradlew` and `gradlew.bat`. This is used for the Java-based web services in `backend/`. It is also used to build Daml smart contracts via a simple wrapper that calls the Daml Assistant[12].

The backend takes advantage of classes generated from the Daml model to simplify interactions with the Ledger API. These are generated directly from the DAR files using the Transcode code generator. The Gradle plugin to run the generator is part of the Transcode package, and is incorporated into the build process in `daml/build.gradle.kts`.

`buildSrc/` contains some custom Gradle plugins in `buildSrc/src/main/kotlin/`:

`ConfigureProfilesTask.kt`	Interactive generation of `.env.local`
`Credentials.kt`	Allows access to credentials stored in `~/.netrc`
`Dependencies.kt`	Propagates version config from `.env` to Gradle
`Repositories.kt`	Adds `digitalasset.jfrog.io` to the Maven artifact repositories
`UnpackTarGzTask.kt`	Provides (required) symlink support for unpacking .tgz files
`VersionFiles.kt`	Provides access .env files and daml.yaml files from Gradle

The project also uses Make[13] as a project choreographer, providing a convenient command-line interface to the various scripts and build tools as well as docker-compose commands. This is similar to the common practice of defining aliases for common dev-loop tasks. Make has the advantage of documenting and sharing these tasks under revision control.[14] Use `make help` to view the currently supported tasks. The `Makefile` itself is intended to be implicit documentation of how each of these steps is performed. By default, `make` also prints any commands it executes to `stdout` and this can also help familiarize new developers to how the dev-loop is structured. If your team is unfamiliar with Make, at the end of this guide[15], we have documented the Make features used in the current Makefile with links to additional documentation.

[12] This wrapper also contains convenience functions to download and install the correct version of the Daml SDK.
[13] https://www.oreilly.com/openbook/make3/book/index.csp
[14] The Makefile is written to be self-documenting, this includes autogenerating "usage" as a default help target
[15] Canton Quickstart Project Structure Short Makefile Primer

Local Deployment (LocalNet) Configuration

Local deployment is handled via Docker[16] and Docker Compose[17] in the usual fashion. Like other blockchains, it constructs a LocalNet on your laptop. In summary:

`.env` and `.env.local` define the necessary environment variables.

`compose.yaml` is the toplevel Docker Compose configuration file

`config/` contains all the various service configuration files required by the various docker containers.

`docker/` contains the various docker image configurations.

LocalNet Port Mappings

For convenience, the LocalNet configuration exposes a number of ports to localhost. For ease of use, the ports are configured using a prefix|suffix arrangement. A single-digit prefix is used to identify the "entity" associated with the relevant node; and, the suffix is the usual four-digit port number associated with the relevant service.

LocalNet Port Prefixes

Prefix	Entity
`2${PORT}`	Application User
`3${PORT}`	Application Provider
`4${PORT}`	Super Validator

LocalNet Port Suffixes

Suffix	Service
`5001`	Participant Ledger API Port
`5002`	Participant Admin API Port
`5003`	Validator Admin API Port
`7575`	Participant JSON API Port
`5432`	Postgres Port

[16] https://docs.docker.com/
[17] https://docs.docker.com/compose/

So for example the `JSON API Port` for the Application User is `27575`; while the `Ledger API Port` for the Super Validator is `25001`.

Important Security Note Regarding Port Mappings

Be aware that the port mappings for LocalNet include exposing both the `AdminAPI` port and the `Postgres` port, both of which would normally be a security risk. Having direct access to these ports when running on a local developers machine can be useful. **These ports should not be exposed when preparing deployment configurations for non-local deployments.**

Should you wish to disable these mappings even for the LocalNet deployment, the port suffixes are defined as environment variables in the `.env`. For any port mappings you wish to disable, you can find and remove the relevant Docker `port:` entry in the `compose.yaml` file.

Application Source

As with most Daml applications the source code falls into four categories:

Application Directories

Directory	Tech Stack	Contents
`daml/`	Daml	The Daml model and DAR dependencies
`frontend/`	React, Vite, Axios, Typescript	Web front end code
`backend/`	Java, Springboot, Protobuf	Back end code. Currently PQS backed OpenAPI endpoints for the front end[18].
`common/`	OpenAPI	Interface definitions shared by one or more of the previous three categories. Currently an openapi.yaml file defining the interface between Front and Back ends

Both the frontend and backend examples can be written using any relevant technology stack. In particular, there is no reason why the backend could not be written using Node.js, C#, or any other language. As of the time this was written, the Daml codegen tooling provided by Digital Asset supports Java, Javascript, and Typescript which has driven the choice of stack for the example application.

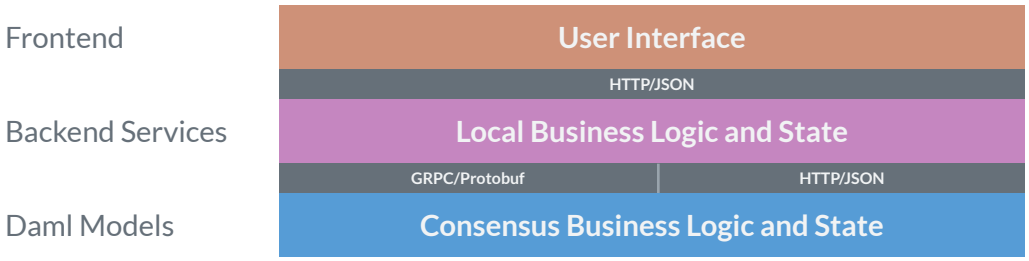
[18] This is also where you should expect to find any automation, integration, and other off-ledger components

Application Structure

Example Application Architecture

It is tempting to see three layers and immediately assume these align with the traditional 3-tier architecture (User Interface, Business Logic, Database), but doing this will result in underperforming applications generating unnecessary traffic on the Global Synchronizer. It is easy to fall into the trap to treat the blockchain as a database because it has very similar features to a database. However, applying standard database design techniques to a blockchain does not produce an optimal design. A better way to view these layers is in terms of consensus vs. local state. Specifically: User Interface, Local Business Logic and State, and Consensus Business Logic and State.

- Local Business Logic and State: Actions and data that a single participant node can handle on their own without needing consensus from others.
- Consensus Business Logic and State: Actions and data that require agreement or validation from multiple parties and need to be handled using Daml smart contracts.



A symptom that you have fallen into the trap of treating the blockchain like a database is a prevalence of CRUD operations in the web-services provided by the backend. The blockchain is intended to synchronize data between multiple organizations with little trust between them. This means that the operations between organizations should be at a larger granularity, invariably representing business operations rather than updates to an object store.

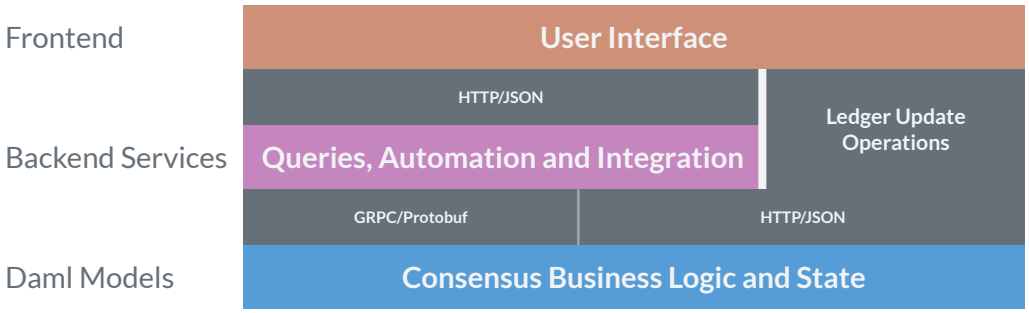
The privacy guarantees provided by Canton do not exist on a publicly visible ledger. So all business logic and business state that need to be either authorized or verified by more than one party is implemented within the Daml smart contract. The necessary consensus on authorization, verification, and/or visibility will then be coordinated via the (Global) Synchronizer. For a more detailed discussion on the distinction between local vs. consensus logic and state see the Daml Philosophy Course 2 "Daml Workflows"[19].

[19] https://daml.talentlms.com/catalog/info/id:152 currently part of the Daml Philosophy Certification https://daml.talentlms.com/catalog/info/id:149

Alternative Application Architecture

This example application could have used a CQRS-style alternative architecture. This architecture is often used where front end user action stories are expressed directly in terms of unmediated consensus business operations. This means:

- User interface updates (writes) are performed directly against the Daml models rather than mediated through backend services.
- User interface queries (reads) remain provided by backend services; which also continue to provide external integrations and automation.



For a detailed discussion on options for application architectures see the free courses in the Technical Solution Architect Certification[20].

Daml Model Structure

```
√ % tree licensing
licensing
├── daml
│   └── Licensing
│       ├── AppInstall.daml
│       ├── License.daml
│       └── Util.daml
└── daml.yaml

3 directories, 4 files
```

The example application is a simple license management application that allows the application

[20] In particular the Solution Topology course https://daml.talentlms.com/catalog/info/id:161 within the larger TSA certification https://daml.talentlms.com/catalog/info/id:160

provider to issue licenses to application users; with license fees paid using Canton Coin. It uses a Daml model consisting of two modules. The `AppInstall` module has two responsibilities:

1. The on-ledger component of user onboarding using the `AppInstallRequest` template
2. The core services provided to each onboarded user through the application using the `AppInstall` template

For the purposes of testing and experimentation there is a make target[21] to create the `AppInstallRequest` on behalf of the app user party `Org1`.

```
.PHONY: create-app-install-request
create-app-install-request: ## Submit an App Install Request from the App ↩
User participant node
	docker compose -f docker/app-user-shell/compose.yaml ↩
$(DOCKER_COMPOSE_ENVFILE) run --rm create-app-install-request || true
```

This uses `curl` via a utility function `curl_check`[22] to submit a Daml `Create` command to Org1's participant node via its HTTP Ledger JSON API (`v2/commands/submit-and-wait`).

```
√ % cat docker/app-user-shell/scripts/create-app-install-request.sh
#!/bin/bash
...
source /app/utils.sh

create_app_install_request() {
  curl_check "http://$participant:7575/v2/commands/submit-and-wait" "$token"
"application/json" \
    --data-raw '{
        "commands" : [
           { "CreateCommand" : {
                "template_id":   "#quickstart-licensing:Licensing.App ↩
Install:AppInstallRequest",
                "create_arguments": {
                    "dso": "'$dsoParty'",
                    "provider": "'$appProviderParty'",
                    "user": "'$appUserParty'",
                    "meta": {"values": []}
                }
```

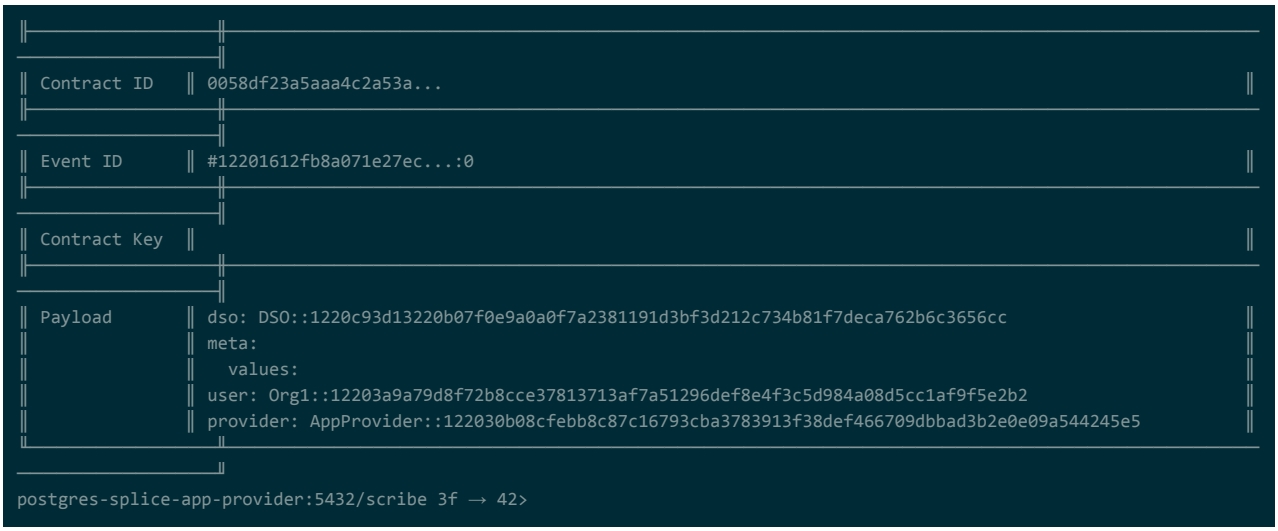
[21] Most make targets can be located by searching/grepping for `^target:`. The main exceptions to this are the `open-*` targets which are cross-platform and generated by macro at the end of the file.
[22] Found in docker/utils.sh

```
              }
             }
            ...


...

create_app_install_request "$LEDGER_API_ADMIN_USER_TOKEN_APP_USER" ↩
$DSO_PARTY $APP_USER_PARTY $APP_PROVIDER_PARTY participant-app-user
```

Running this and then using Daml Shell[23] (`make shell` provides a useful shortcut) to inspect the result on the ledger.

```
√ % make shell
docker compose -f docker/daml-shell/compose.yaml --env-file .env run --rm daml-shell || true
Connecting to jdbc:postgresql://postgres-splice-app-provider:5432/scribe...
Connected to jdbc:postgresql://postgres-splice-app-provider:5432/scribe
postgres-splice-app-provider:5432/scribe> active
┌──────────────────────────────────────────────────────────────────────┬──────────┬─────────┐
│ Identifier                                                           │ Type     │ Count │
╞══════════════════════════════════════════════════════════════════════╪══════════╪═══════╡
│ quickstart-licensing:Licensing.AppInstall:AppInstallRequest          │ Template │     1 │
├──────────────────────────────────────────────────────────────────────┼──────────┼───────┤
│ splice-amulet:Splice.Amulet:ValidatorRight                           │ Template │     1 │
├──────────────────────────────────────────────────────────────────────┼──────────┼───────┤
│ splice-wallet:Splice.Wallet.Install:WalletAppInstall                 │ Template │     1 │
└──────────────────────────────────────────────────────────────────────┴──────────┴───────┘
postgres-splice-app-provider:5432/scribe 3f → 42> active quickstart-licensing:Licensing.AppInstall:AppInstallRequest
┌────────────┬────────────┬────────────┬──────────────────────────────────────────────────────────────────────────────────────────────────────────┐
│ Created at │ Contract ID │ Contract Key │ Payload                                                                                                  │
╞════════════╪════════════╪════════════╪══════════════════════════════════════════════════════════════════════════════════════════════════════════╡
│ 42         │ 0058df23a5aaa4c2a53... │ dso: DSO::1220c93d13220b07f0e9a0a0f7a2381191d3bf3d212c734b81f7deca762b6c3656cc                            │
│            │                        │ meta:                                                                                                    │
│            │                        │   values:                                                                                                │
│            │                        │ user: Org1::12203a9a79d8f72b8cce37813713af7a51296def8e4f3c5d984a08d5cc1af9f5e2b2                          │
│            │                        │ provider: AppProvider::122030b08cfebb8c87c16793cba3783913f38def466709dbbad3b2e0e09a544245e5              │
└────────────┴────────────┴────────────┴──────────────────────────────────────────────────────────────────────────────────────────────────────────┘
postgres-splice-app-provider:5432/scribe 3f → 42> contract
0058df23a5aaa4c2a53aab496d12fb9e8ee74fb91614e5f7d50670598e4760eb23ca101220cc241620b310c93af45b2cd7cea7518e18e26f73f227813
fec2bf4ea0bd69b94
╔═══════════════╦═══════════════════════════════════════════════════════════════════════════════════════════════════════════════════════════════╗
║ Identifier    ║ quickstart-licensing:Licensing.AppInstall:AppInstallRequest                                                                   ║
╠═══════════════╬═══════════════════════════════════════════════════════════════════════════════════════════════════════════════════════════════╣
║ Type          ║ Template                                                                                                                      ║
╠═══════════════╬═══════════════════════════════════════════════════════════════════════════════════════════════════════════════════════════════╣
║ Created at    ║ 42 (not yet active)                                                                                                           ║
╠═══════════════╬═══════════════════════════════════════════════════════════════════════════════════════════════════════════════════════════════╣
║ Archived at   ║ <active>                                                                                                                      ║
```

[23] https://docs.daml.com/tools/daml-shell/index.html#daml-shell-daml-shell



Exercising the `AppInstallRequest_Accept` choice completes the onboarding. The Frontend UI provides a way to do this.

Key Daml Templates

AppInstallRequest Contract

The `AppInstallRequest` contract initiates the app user onboarding process by capturing a user's request to install the application. The contract gives the application provider (henceforth just *provider*) control over application access to accept or reject installation requests. This contract offers three choices that extend the Propose/Accept pattern[24] to allow the user to cancel the request.

The `AppInstallRequest_Accept` choice allows the provider to accept the request. When the choice is executed, it creates a new AppInstall contract and makes the provider and user signatories.

The `AppInstallRequest_Reject` choice allows the provider to decline the request. It archives the request contract and also records in the ledger exercise event, metadata about why the request was rejected.

The `AppInstallRequest_Cancel` choice allows the user to withdraw their request any time before the provider accepts the contract.

[24] https://docs.daml.com/daml/patterns/propose-accept.html

AppInstall Contract

The `AppInstall` contract maintains the formal relationship between the provider and user. It tracks installation status and manages license creation. The contract has two choices, `AppInstall_CreateLicense` and `AppInstall_Cancel`.

`AppInstall_CreateLicense` allows the provider to create a new license for the user. When the `CreateLicense` choice is exercised it creates a new `License` contract. It also increments `numLicensesCreated` to track how many licenses exist which is used to assign each licence a licence number. **Note:** Daml smart contracts are immutable, so "incrementing" the counter results in archiving the current `AppInstall` contract and creating a new one with the updated counter, within the same atomic transaction.

`AppInstall_Cancel` lets the provider or user cancel the installation.

License Contract

The `License` contract is the on ledger record supporting the core business case for the application. One critical field is the `expiresAt` field, which both determines the duration of the license's validity, and is used to ensure that neither actor can revoke (ie. archive) the license contract before expiry. The contract also has two choices:

`License_Renew` can be exercised by the license provider. It creates a Splice[25] `AppPaymentRequest` and a `LicenseRenewalRequest` contract. The former is a part of the Splice Wallet Application, and is used to request an amulet transfer. The choice of which amulet is made via the `dso` party used in the `AppInstall` contract. The current deployment configuration will result in this being Canton Coin; however, there is nothing in the Daml model, or the backend code that prevents a different amulet being used.

The `License_Expire` choice allows either party to archive an expired `License` contract. This has the benefit of allowing an expired license to be renewed without having to reissue it. It is also necessary because Daml smart contracts do not have any facility to self-execute or self-archive. Every change to the ledger originates from a command submitted to the ledger API on a validator. As a result this sort of cleanup operation must be exercised explicitly via a choice such as this. It is not uncommon to have background or end-of-day batch processes automate this sort of task.

Common OpenAPI Definition

The Daml models define the consensus between the App Provider, App User, and the DSO (amulet issuer). Once the models are in use, the front end user interface needs to be able to query and interact with the resulting ledger. The usual pattern is to store and index the relevant slice of the

[25] https://docs.sync.global/index.html

ledger in the Participant Query Store[26], and provide a set of query web services that provide business oriented queries resolved against the PQS postgres database.

The architecture used by the example application also exposes a variety of HTTP endpoints that allow the frontend to exercise choices, providing a bridge between the frontend and the GRPC Ledger API. This allows the backend to centralise authentication and access control code.

This does necessitate defining an API between the back and front ends. For this example application, we have chosen to use OpenAPI[27]. The API definition is in `common/openapi.yaml`. It uses `GET` to access the query services in the backend; and, `POST` to execute choices on contracts identified by contract-id in the URL.

Note: This is using HTTP. The HTTP method semantics align appropriately with the requirements of the Daml operations and we call this a "JSON API". However, it is not a pure ReST[28] API and does use HATEOAS. As mentioned above, the blockchain should not be viewed as a database since the underlying state is not rows in a database, or objects in a datastore—either of which would be compatible with the CRUD-style semantics that emerge with most modern ReST tooling. Instead the architecture style used here is more akin to a sophisticated RPC mechanism[29].

Backend Services Structure

The example backend is a SpringBoot[30] application the core of which are the API implementation classes in `com.digitalasset.quickstart.service`.

Most of this code is standard Java SQL-backed JSON-encoded HTTP web service fare. The code itself is divided into seven modules under `com.digitalasset.quickstart.*`:

`config`: Mostly standard SpringBoot `@ConfigurationProperties` based components; however, `SecurityConfig` may be worth looking at for how the example application handles CSRF tokens and OAuth2 authentication of login and logout requests.

`oauth`: Amongst other things, provides a client interceptor to authenticate the backend services to the Ledger API.

[26] https://docs.daml.com/query/pqs-user-guide.html#pqs
[27] https://www.openapis.org/
[28] As defined by Roy Fielding (https://ics.uci.edu/~fielding/pubs/dissertation/top.htm)
[29] Contract-ids and their underlying contract are nouns and can be represented as ReST resources. However, not only does this fail to capture the ongoing business entity that often outlives any single contract, it misses the fact that at the core of Daml are the authorized choices which are verbs and therefore do not play nicely with ReST assumptions.
[30] https://spring.io/projects/spring-boot

`service`: Implements the openAPI endpoints. Mostly a roughly equal split between read-only calls to PQS via the `DamlRepository` spring component; and, GRPC calls to the relevant validator via the `LedgerApi` spring component.

`ledger`: The main class here is `LedgerApi` which handles the details of calling the relevant GRPC endpoints required to submit Daml commands and other requests to the Canton Validator.

`repository`: Includes `DamlRepository`. A `@Repository` component providing business-logic level query and retrieval facilities against the ledger via PQS (the Participant Query Store).

`pqs`: The main class is `Pqs`, which provides data-model level query and retrieval. This encapsulates the necessary SQL generation, and the JDBC queries against the PQS Postgres database.

`utility`: For the moment this is restricted to the `ObjectMapper` required for JSON transcoding in the web services.

Ultimately the main recommendation embedded in this code is to orient the web-service api around a combination of queries and choice invocations. This is hopefully adequately demonstrated in the open API definition. Other than that the usual web service engineering considerations apply. Separation of concerns, DRY[31], and the importance of centralising SQL generation and Authentication mechanisms to ensure having to address these security sensitive components only once.

Frontend Interface Structure

One property of the fully mediated architecture used in the example application is that by delegating all operations to the backend, the open API schemas act as DTO (Data Transfer Object) definitions for the front and back ends[32]. In simple cases, such as the example application, these can double as front end models when using a React, MVVM, FRP, or similar front end architecture style.

The example application is a naive React web frontend[33] written in Typescript[34]. It accesses the Backend web services using the generator-less Axios client to handle the lowest level transport, configured in `src/api.ts`:

[31] Topic 9 https://pragprog.com/titles/tpp20/the-pragmatic-programmer-20th-anniversary-edition/ "Don't Repeat Yourself"
[32] The CQRS alternative architecture does not use DTOs. Instead the backend services return Daml contracts directly. These are then generally deserialised directly into Javascript or Typescript objects, generated directly from the DAR files; and, used to populate the underlying frontend model. This direct coupling from Daml to Frontend can significantly simplify the code required for applications with requirements defined in terms of a Daml model. The mediated architecture is more suitable where the Frontend needs to incorporate sources of data additional to the Canton Ledger.
[33] https://react.dev/
[34] https://www.typescriptlang.org/

```
import OpenAPIClientAxios from 'openapi-client-axios';
import openApi from '../../common/openapi.yaml'

const api = new OpenAPIClientAxios({
    definition: openApi as any,
    withServer: { url: '/api' },
});

api.init();

export default api;
```

Authentication is handled using OAuth2 against a mock OAuth server[35] to perform the login; and, bearer tokens to identify the Frontend to the Backend. The Frontend does not have any knowledge of Canton, or Daml Users or Parties, this is delegated entirely to the Backend.

The records defined by the OpenAPI definition are used directly as the models maintained within the react stores, and from there to the views via the usual react handlers.

Short Makefile Primer

Make is the original build tool developed to assist with C development on UNIX in 1976[36]. As such it relies heavily on transparent integration with the unix shell. To this day Make retains the most comprehensive and seamless shell integration of any build tool available — which is why it makes a good choreography tool. The version used in this project is GNU Make[37], which has a number of useful extensions.

The basic format of a make build target is:

```
.<SPECIAL-TARGET-DECLARATIONS>: <target-name>
<target>: <dependency list (space separated)>
      shell commands, make macros, and gnu-make function invocations
```

For instance to build the front-end you can run `npm install && npm run build` from the `frontend/` directory; or, `make build-frontend` from the `quickstart/` directory via the following target in `quickstart/Makefile`:

[35] This is being changed to use keycloak as the JST server.
[36] https://en.wikipedia.org/wiki/Make_(software)
[37] https://www.gnu.org/software/make/manual/html_node/index.html

```
.PHONY: build-frontend
build-frontend: ## Build the frontend application
	@cd frontend && npm install && npm run build
```

`.PHONY`[38] is a special built-in target that is used to indicate that `build-frontend` is strictly a target name and does not correspond to a file

`build-frontend`: Is a build target which can be invoked directly via `make <target>` or indirectly as a dependency for another target. If not marked as a phony-target it will be treated as a file, and the last-modified timestamp compared to its dependencies in the usual manner.

`#` Is a line comment delimiter, identical to a shell script.

`##` is not a Make concept, but is used by convention as a doc-string to generate the usage displayed by `make help`.

`<tab>@cd frontend && npm install && npm run build` is a shell command to be executed when the target is invoked. Unless this is a phony-target, the expectation is that this command will regenerate the target file. By default `make` prints each shell command to stdout immediately before it executes it, this is suppressed if the command is prepended with a `@`.

***NOTE:** The shell-command **MUST** be indented by a literal **TAB** character, the equivalent number of spaces **WILL NOT WORK**.*

You can see dependency list in action with the top-level `build` target:

```
.PHONY: build
build: build-frontend build-backend build-daml build-docker-images
```

When the target is invoked the dependency targets are run brought up to date (ie. in invoked in the case of phony targets) before any shell command is executed.

Other Make features that are currently used in the existing file include:

`define`[39] which is used to define multiline variables. In this case to define a simple macro (`open-url-target`) to define crossplatform browser interaction targets (try `make open-app-ui` once the application is started for an example). The file also includes:

[38] https://www.gnu.org/software/make/manual/html_node/Phony-Targets.html
[39] https://www.gnu.org/software/make/manual/html_node/Multi_002dLine.html

```
# Function to run docker-compose with default files and environment
define docker-compose
        docker compose $(DOCKER_COMPOSE_FILES) $(DOCKER_COMPOSE_ENVFILE) ↩
$(DOCKER_COMPOSE_PROFILES) $(1)
endef
```

This provides DRY abstraction around calls to `docker-compose`.

`call`[40] which is used to invoke a variable as a function.

Note the format of a `call` invocation is: `$(call <cmd>[, <args>]*)`. So `$(call open-url-target, open-app-ui, http://localhost:3000)` calls `open-url-target` with `$(1)` set to the string `open-app-ui` and `$(2)` set to the url.

Similarly, the `make status` target uses `$(call docker-compose, ps)` to run `docker-compose ps` with the default arguments. This happens via the `docker-compose` function discussed above. Removing the `@` will allow you to see the expanded command. ie.

```
√ % make status
docker compose -f compose.yaml --env-file .env --profile localnet --env-file
docker/localnet.env --profile observability  ps
```

`eval`[41] which is used to treat the result of calling `open-url-target` as a macro to define dynamic make targets.

[40] https://www.gnu.org/software/make/manual/html_node/Call-Function.html
[41] https://www.gnu.org/software/make/manual/html_node/Eval-Function.html