Backend Authentication

Canton Network Quickstart Guide | 2025

Version: 0.9.0-2025-04-22

Contents

Authentication Overview

General Architecture

Canton Network components are designed to delegate end-user identity and authentication to external IAM services run by each participant. In practice this means using OAuth to secure access to the ledger[1]. The Example Application therefore takes the logical step of extending this same principle to the Backend Service, and indeed to any other components that require secure access control.

Consequently there are three concepts of identity that need to be reconciled in a Canton Network Application.

Public **Ledger Identity**[2] is encoded as **Daml Parties**[3]. These correspond to ledger key-pairs installed and/or created on a Participant Node using the Canton Console[4]; and, are the only explicit encoding of identity on the blockchain.

It is recommended that access to the **Ledger API** be configured in terms of **Participant Users**. These represent a stable local identity that can be configured with readAs or actAs (ie. read/write) access to Daml Parties hosted on the same Participant Node. It is expected that the identity of each application/service and end-user that requires *direct* access to the ledger API will be encoded as a participant user.

IAM Users provide the third encoding of identity. These represent an entity that is capable of authenticating itself to the application. Canton does not provide any direct support for creating or managing these users, as it is expected that any Canton Network participant will have their own pre-existing IAM or Directory services. Instead Canton defines **Access Token Formats** to be issued by the IAM that provide authorization to access either Participant-Users or direct access to Daml Parties.

The Access Token Formats are **JSON Web Tokens (JWTs)**. They follow the OAuth 2.0 standard, and Canton uses various **OAuth2** authorization flows to validate any tokens it receives. Access Tokens come in two broad categories:

User Access Tokens encode authorization to access a participant user on a Participant Node.

Custom Daml Claims Access Tokens encode the rights normally assigned to a Participant User directly in the token.

[1] https://docs.daml.com/app-dev/authorization.html#authorization
[2] https://docs.daml.com/canton/usermanual/identity_management.html
[3] https://docs.daml.com/concepts/glossary.html#party
[4] https://docs.daml.com/canton/reference/console.html#party-management

Example Application Overview

Recall a Canton Network Validator is a Canton Participant Node that is also running the Canton Coin Wallet application. These various components use **OpenID Connect (OIDC)**[5] as the authentication protocol to the relevant IAM. The example application includes **KeyCloak** to provide a local IAM to support Local Net deployments.

For documentation on configuring a validator node to use an OIDC provider see the authentication sections in the Splice documentation.[6] A Docker-based deployment, such as Local Net, is configured via environment variables. These are normally configured via the `.env` file; however, due to the example application including multiple validators and a super-validator, there is a two step process where groups of variables are configured in the `.env` and then assigned to the specific configuration variables in the relevant Docker `compose.yaml` files.

For instance for the Application Provider, the `.env` file (at the time of writing) contains:

Application Provider Authentication Configuration

Environment Variable	Value
AUTH_APP_PROVIDER_ISSUER_URL	http://keycloak.localhost:8082/realms/AppProvider
AUTH_APP_PROVIDER_ISSUER_URL_BACKEND	http://nginx-keycloak:8082/realms/AppProvider
AUTH_APP_PROVIDER_WELLKNOWN_URL	${AUTH_APP_PROVIDER_ISSUER_URL_BACKEND}/.well-known/openid-configuration
AUTH_APP_PROVIDER_TOKEN_URL	${AUTH_APP_PROVIDER_ISSUER_URL_BACKEND}/protocol/openid-connect/token
AUTH_APP_PROVIDER_JWK_SET_URL	${AUTH_APP_PROVIDER_ISSUER_URL_BACKEND}/protocol/openid-connect/certs
AUTH_APP_PROVIDER_AUDIENCE	https://canton.network.global
AUTH_APP_PROVIDER_PARTY_HINT	app_provider_${PARTY_HINT}
AUTH_APP_PROVIDER_BACKEND_CLIENT_ID	app-provider-backend
AUTH_APP_PROVIDER_BACKEND_SECRET	05dmL9DAUmDnIlfoZ5EQ7pKskWmhBlNz
AUTH_APP_PROVIDER_BACKEND_USER_ID	1a36eb86-4ccc-4ec6-b7b7-caa08b354989
AUTH_APP_PROVIDER_BACKEND_OIDC_CLIENT_ID	app-provider-backend-oidc

[5] https://openid.net/developers/how-connect-works/
[6] https://docs.test.global.canton.network.sync.global/validator_operator/validator_helm.html#configuring-authentication and https://docs.test.global.canton.network.sync.global/validator_operator/validator_compose.html#configuring-authentication

Application Provider Authentication Configuration

Environment Variable	Value
AUTH_APP_PROVIDER_VALIDATOR_CLIENT_ID	app-provider-validator
AUTH_APP_PROVIDER_VALIDATOR_CLIENT_SECRET	AL8648b9SfdTFImq7FV56Vd0KHifHBuC
AUTH_APP_PROVIDER_VALIDATOR_USER_ID	c87743ab-80e0-4b83-935a-4c0582226691
AUTH_APP_PROVIDER_WALLET_UI_CLIENT_ID	app-provider-wallet
AUTH_APP_PROVIDER_WALLET_ADMIN_USER_ID	553c6754-8879-41c9-ae80-b302f5af92c9
AUTH_APP_PROVIDER_PQS_CLIENT_ID	app-provider-pqs
AUTH_APP_PROVIDER_PQS_CLIENT_SECRET	zuYvMzWEo8csYNiQNImXNPsmPErBWP3W

Compare this with the environment section of the `validator-app-provider` service configuration in `compose.daml`:

```
environment:
  - *canton-metrics-config
  - ONBOARDING_SECRET_URL=${ONBOARDING_SECRET_URL}
  - SPLICE_APP_VALIDATOR_PARTICIPANT_ADDRESS=${APP_PROVIDER_VALIDATOR_PARTICIPANT_ADDRESS}
  - SPLICE_APP_VALIDATOR_LEDGER_API_AUTH_CLIENT_ID=${AUTH_APP_PROVIDER_VALIDATOR_CLIENT_ID}
  - SPLICE_APP_VALIDATOR_LEDGER_API_AUTH_CLIENT_SECRET=${AUTH_APP_PROVIDER_VALIDATOR_CLIENT_SECRET}
  - SPLICE_APP_VALIDATOR_LEDGER_API_AUTH_AUDIENCE=${AUTH_APP_PROVIDER_AUDIENCE}
  - SPLICE_APP_VALIDATOR_LEDGER_API_AUTH_USER_NAME=${AUTH_APP_PROVIDER_VALIDATOR_USER_ID}
  - SPLICE_APP_VALIDATOR_WALLET_USER_NAME=${AUTH_APP_PROVIDER_WALLET_ADMIN_USER_ID}
  - SPLICE_APP_VALIDATOR_AUTH_AUDIENCE=${AUTH_APP_PROVIDER_AUDIENCE}
  - SPLICE_APP_VALIDATOR_AUTH_JWKS_URL=${AUTH_APP_PROVIDER_JWK_SET_URL}
  - SPLICE_APP_VALIDATOR_LEDGER_API_AUTH_URL=${AUTH_APP_PROVIDER_WELLKNOWN_URL}
  - SPLICE_APP_VALIDATOR_PARTY_HINT=${AUTH_APP_PROVIDER_PARTY_HINT}
  - |
    ADDITIONAL_CONFIG_PERSISTENCE=
        canton.validator-apps.validator_backend.storage {
          type = postgres
          config {
            dataSourceClass = "org.postgresql.ds.PGSimpleDataSource"
            properties = {
              databaseName = validator
              currentSchema = validator
              serverName = ${SPLICE_APP_PROVIDER_DB_SERVER}
              portNumber = ${SPLICE_APP_PROVIDER_DB_PORT}
              user = ${SPLICE_APP_PROVIDER_DB_USER}
              password = ${SPLICE_APP_PROVIDER_DB_PASSWORD}
            }
          }
        }
```

Authentication in the Backend

One advantage of the backend mediated architecture utilized by the example application is that it centralises the ledger authentication concerns within a single component. JWT bearer credentials are required for authentication to a Canton Participant or Validator[7] Node. Obtaining, securing, and properly managing these credentials invariably requires some form of framework support. Centralising participant access means:

1. Only one authentication framework to understand and configure
2. Only one place where sensitive ledger credentials are stored and/or accessed
3. A single point of responsibility where finer grained authorization rules or privacy filters can be deployed beyond the public consensus rules defined on-ledger in the Daml models

As a Java-based Spring application, the example application uses Spring Security to provide the necessary framework for managing authentication. The rest of this section describes the various elements of that deployment—the considerations covered here will need to be addressed by whichever language and framework combination you ultimately chose to use.

While looking at the backend authentication keep in mind the following:

1. **IAM Users** are hosted by your chosen user directory or identity access management service, and authenticate individual **local end-users** or **services**
2. **Parties** represent **on-ledger identity** and are hosted by Participant/Validator Nodes
3. **Participant Users** represent a stable local identity that can be configured with authorization to access parties

Configuring the Backend Service Users

Daml Parties are associated with key-pairs and UUID's that are generated upon party creation and installation in a PN (Participant Node). Similarly, Participant Users also need to be created and configured on a running PN. Because of this, the Party and User configurations cannot be defined statically in the example application `config/` directory, but instead have to be installed using various bootstrap scripts shortly after the initial deployment. All this happens transparently through the docker-compose invocation when running `make start`.

[7] Recall a Canton Validator Node is a Canton Participant Node that is also running the necessary Splice Wallet application required to synchronize transactions using the Global Synchronizer (https://www.canton.network/global-synchronizer).

Corresponding to the three elements mentioned above:

IAM Users

As mentioned, the example application does not use mock authentication, but instead provides a fully fledged OAuth management service in KeyCloak. IAM Users have been statically defined using the KeyCload admin interface, and then serialized in `docker/oauth/data`. These define two realms (AppProvider and AppUser) containing multiple users as detailed in `docker/oauth/data/README.md`. For the purposes of the backend the key users are:

- `app-provider-backend`
- `app-provider-pqs`
- `App-provider-backend-oidc`

Parties

The default Canton Validator deployment creates a party automatically based on a PARTY_HINT configured via `.env.local` and either `env/app-provider.env` or `env/app-user.env`; whichever is appropriate. As mentioned, because this party only exists once the validator bootstrap process is complete, the rest of the authentication configuration must also be dynamic.

Participant Users

The participant users are configured via a dedicated docker container: `splice-auto-config`. In both cases the users are created and granted the appropriate rights via the health check script (latched via a lock file). For the purposes of the backend, the key file is the `app-provider.sh` script, but for details of how precisely users are created, configured, and associated with ledger parties see the relevant functions in `docker/utils.sh`.

- `create_user` which POST's a create user request to the Admin API `http://$participant/v2/users`; and,
- `grant_rights` which POST's the requested authorizations to the requested parties via `http://$participant/v2/users/$userId/rights`

It is worth checking these shell functions out, as well as the `curl_check` and `curl_status_code` for an example of how to interact with ledger JSON APIs — including injecting the necessary authorization bearer tokens.

Backend Security Configuration

The security relevant sections of the `application.yml` file is included below for reference.

```
security:
  issuer-url: ${AUTH_APP_PROVIDER_ISSUER_URL}

application:
  tenants:
    AppProvider:
      tenantId: AppProvider
      partyId: ${AUTH_APP_PROVIDER_PARTY}
      internal: true
```

These are used to populate the usual ConfigurationProperties beans. The key properties here that represent the minimal configuration for a ledger client are:

- `tenantId` the identity of the backend service within the local IAM service
- `issuer-url` the identity of the local IAM service
- `partyId` the on-ledger Daml Party associated with the tenant

This last can be relevant when querying PQS as the query store is accessed in terms of ledger identity (ie party) not local participant identity (participant user). In the case of the example application the PQS configuration utilizes an `AUTH_APP_PROVIDER_PQS_USER_NAME`[8] configured with `ReadAs` rights to the `partyId` to populate the PQS database

The `issuer-url/issuer-uri` is a used both both by the backend to verify received JWT's are issued by the correct issuer; and, by the Spring Framework as part of various OAuth credential flows.

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: ${security.issuer-url}

      client:
        registration:
          AppProviderBackend:
            client-id: ${AUTH_APP_PROVIDER_BACKEND_CLIENT_ID}
            client-secret: ${AUTH_APP_PROVIDER_BACKEND_SECRET}
            authorization-grant-type: client_credentials
```

[8] Currently configured in `env/app-provider.env` and set to `service-account-app-provider-pqs`.

```
            provider: AppProvider
          AppProvider:
            client-id: ${AUTH_APP_PROVIDER_BACKEND_OIDC_CLIENT_ID}
            client-name: ${application.tenants.AppProvider.tenantId}
            authorization-grant-type: authorization_code
            scope: "openid"
            redirect-uri: "{baseUrl}/login/oauth2/code/{registrationId}"
            provider: AppProvider
        provider:
          AppProvider:
            issuer-uri: ${security.issuer-url}
```

Being a Java-based SpringBoot Application the example application naturally uses Spring Security as its authentication framework. The above configuration should be read as such, and the documentation for the Spring Security Authentication Architecture[9], and for the OAuth2[10] implementation are comprehensive.

If you are using another framework note that the above configuration includes configurations for both the Backend as an OAuth service client — in which capacity it needs to authenticate against PQS and the Validator Node; and as an OAuth service provider, authenticating end users connecting from the Frontend.

Backend to Validator

Authentication is one of several cross-cutting concerns when interacting with the Canton Validator. Connection management, flow-control, error handling, tracing, as well as authentication all mean you should expect to centralise access to each API in a suitable adapter class[11]. In the case of the example application these are `com.digitalasset.quickstart.ledger.LedgerApi` and `com.digitalasset.quickstart.ledger.ScanProxy`.

LedgerAPI (Authenticating with the GRPC binding)

The Participant Ledger API is defined as a GRPC interface using protobuf[12]. While this API is also available via an openapi JSON/HTTP interface, the example application uses the Java GRPC bindings.

[9] https://docs.spring.io/spring-security/reference/servlet/authentication/architecture.html
[10] https://docs.spring.io/spring-security/reference/servlet/oauth2/index.html
[11] Or component, module, interface, or equivalent in your preferred language/tech-stack.
[12] (need to find this)

```
    public LedgerApi(LedgerConfig ledgerConfig, TokenProvider tokenProvider) {

        ManagedChannel channel = ManagedChannelBuilder
                .forAddress(ledgerConfig.getHost(), ledgerConfig.getPort())
                .usePlaintext()
                .intercept(new Interceptor(tokenProvider))
                .build();
```

All invocations of the LedgerAPI require an Authorization header with a suitable bearer token. In the case of the Java GRPC bindings this is achieved using an interceptor that obtains and injects the token.

```
private static class Interceptor implements ClientInterceptor {
  private final Metadata.Key<String> AUTHORIZATION_HEADER =
      Metadata.Key.of("Authorization", Metadata.ASCII_STRING_MARSHALLER);
  private final TokenProvider tokenProvider;

  public Interceptor(TokenProvider tokenProvider) {
    this.tokenProvider = tokenProvider;
  }

  @Override
  public <ReqT, RespT> ClientCall<ReqT, RespT> interceptCall (
      MethodDescriptor<ReqT, RespT> method,
      CallOptions callOptions,
      Channel next) {

    ClientCall<ReqT, RespT> clientCall = next.newCall(method, callOptions);

    return new ForwardingClientCall
        .SimpleForwardingClientCall<ReqT, RespT>(clientCall) {
      @Override
      public void start(Listener<RespT> responseListener, Metadata headers) {
        Headers.put(AUTHORIZATION_HEADER, "Bearer " + tokenProvider.getToken());
        super.start(responseListener, headers
      }
    };
  }
}
```

The specifics of how tokens are generated are specific to the framework you use for your backend;

however, the example application wraps the Spring Security Framework's implementation[13] in the AuthService `@Component` and the SpringSecurityConfig `@Configuration`. These are autowired and autoconfigured from the `application.yml` file discussed above.

ScanAPI (Authenticating with HTTP)

The Canton Validator's Scan API is accessed via an OpenAPI HTTP/ReST endpoint. Again, access to openapi service interfaces is language specific. In the case of Java, authorization is handled via the ApiClient interface by injecting an interceptor into the request builder pipeline. This is done in `ScanProxyApiConfiguration`:

```java
@Configuration
public class ScanProxyApiConfiguration {

    @Bean
    public ScanProxyApi scanProxyApi(
                TokenProvider tokenProvider,
                LedgerConfig ledgerConfig) {

        ApiClient apiClient = new ApiClient();
        apiClient.updateBaseUri(ledgerConfig.getValidatorUri());
        apiClient.setRequestInterceptor(requestBuilder -> {
            requestBuilder.header("Authorization",
                    "Bearer " + tokenProvider.getToken());
        });

        return new ScanProxyApi(apiClient);
    }
}
```

If you aren't using openapi generated bindings then you will need to add the `Authorization` header using whichever Http Request api you are using. Keep in mind that all APIs on the Canton Participant/Validator nodes expect bearer tokens using JWT encoded OAuth2 credentials.

Frontend to Backend

Authentication and authorization to the Canton Ledger are unavoidably dictated by the requirements of the Participant Node and Validator application APIs. This does not apply to authentication of Frontend applications to your Backend. Here you have a free choice, the only

[13] https://docs.spring.io/spring-security/reference/servlet/oauth2/client/index.html

requirement is that you be able to map each authenticated frontend request to an appropriate Participant credential (ie. a suitable JWT).

As the example application already runs KeyCloak as its IAM to provide OAuth support for the Canton components, the Front and Back ends use their own Realm in KeyCloak to configure and authenticate front end users during Local Net and tests. This does mean that the Backend uses the Spring Security OAuth2 Resource Server support. For more details see Spring's documentation[14].

The example application uses the "override auto configuration" option in order to setup CSRF, permit access to login pages, success/failure handlers, and some sanity cleanup on logout. This is provided by the filterChain `@Bean` in SpringSecurityConfig. Note that the association between user and ledger credentials is the core of the `onAuthenticationSuccess` handler in OAuth2AuthenticationSuccessHandler:

```
Map<String, Object> claimsWithParty = new HashMap<>(oidcUser.getClaims());
claimsWithParty.put(AuthService.VIRTUAL_TENANT_ID_CLAIM,
    clientReg.getClientName());
claimsWithParty.put(AuthService.VIRTUAL_PARTY_ID_CLAIM,
    tenantPropertiesRepository.getTenant(
        clientReg.getClientName()).getPartyId());

OidcIdToken idTokenWithParty = new OidcIdToken(
    oidcUser.getIdToken().getTokenValue(),
    oidcUser.getIssuedAt(),
    oidcUser.getExpiresAt(),
    claimsWithParty);

OAuth2AuthenticationToken newAuth = new OAuth2AuthenticationToken(
    new DefaultOidcUser(authorities, idTokenWithParty, oidcUser.getUserInfo()),
    authorities,
    auth.getAuthorizedClientRegistrationId());

SecurityContextHolder.getContext().setAuthentication(newAuth);
```

Backend to PQS

The interface to PQS is via JDBC, and the current example application uses basic username - password based authentication. This is configured in the JdbcDataSource class configured via the PostgresConfig `@Component`.

[14] https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/jwt.html

```java
@Bean
public DataSource dataSource() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName("org.postgresql.Driver");
    String url = String.format("jdbc:postgresql://%s:%d/%s",
            postgresConfig.getHost(), postgresConfig.getPort(),
            postgresConfig.getDatabase());
    logger.info("Connecting to {} as {}", url, postgresConfig.getUsername());
    dataSource.setUrl(url);
    dataSource.setUsername(postgresConfig.getUsername());
    dataSource.setPassword(postgresConfig.getPassword());
    return dataSource;
}
```