Multiple Authentication Providers

Version 1.3.0-SNAPSHOT, 2026-02-23T21:42:22Z

Table of Contents

Quantum supports running multiple authentication providers simultaneously, enabling scenarios such as an internal JWT provider alongside an external OIDC provider (e.g., Cognito, Keycloak, Auth0). Providers are registered via a single configuration property, and the framework automatically routes token validation to the correct provider based on the JWT `iss` (issuer) claim.

Chapter 1. How It Works

The `AuthProviderFactory` is the central registry for all configured authentication providers. At startup it reads the `auth.provider` property, which accepts a comma-separated list of provider names, instantiates each one via CDI, and holds them in an ordered list.

```
# Single provider (default)
auth.provider=custom

# Multiple providers
auth.provider=custom,cognito
```

1.1. Provider Resolution

When a request arrives with a JWT, the framework resolves which provider should validate it:

1. **By issuer** — `AuthProviderFactory.getProviderForIssuer(String issuer)` iterates through all configured providers and calls `getIssuer()` on each. The first provider whose issuer matches the JWT `iss` claim wins.
2. **By name** — `AuthProviderFactory.getProviderByName(String name)` performs a case-insensitive lookup (e.g., `"custom"`, `"cognito"`).
3. **Default** — `AuthProviderFactory.getAuthProvider()` returns the first configured provider. If issuer-based lookup finds no match, the default provider is used as a fallback.

1.2. Canonical Identity Service

`CanonicalIdentityService` is the single entry point for token validation across all providers. It has two signatures:

```
// Use the default provider
SecurityIdentity validateAccessTokenCanonical(String token);

// Route to the provider matching this issuer
SecurityIdentity validateAccessTokenCanonical(String token, String issuer);
```

Internally, the service supports two provider interfaces:

- **`ClaimsAuthProvider`** — a modern SPI where the provider returns a `ProviderClaims` object (subject, roles, attributes). The service then passes those claims to `IdentityAssembler`, which resolves the userId from the credential store, merges roles from the token, credential, and user groups, and builds a canonical `QuarkusSecurityIdentity`.
- **Legacy `AuthProvider`** — the provider returns a `SecurityIdentity` directly, and the service normalizes it via `SecurityIdentityNormalizer` (ensuring the principal is always the userId, not the IdP subject).

This means you can mix modern claims-based providers with legacy ones in the same application.

Chapter 2. Configuring `application.properties`

2.1. Minimal Custom JWT Provider

```
# Select the custom JWT provider
auth.provider=custom

# Enable SmallRye JWT validation
quarkus.smallrye-jwt.enabled=true

# Public key for token signature verification (classpath resource)
mp.jwt.verify.publickey.location=publicKey.pem

# Issuer claim that tokens must contain
mp.jwt.verify.issuer=https://myapp.example.com/issuer

# Audiences the token must target
mp.jwt.verify.audiences=my-api-client,my-api-client-refresh

# Signing secret for the custom provider (used for HMAC-based flows)
auth.jwt.secret=${JWT_SECRET:change-me-in-production}

# Token lifetimes (minutes)
auth.jwt.expiration=15
auth.jwt.refresh-expiration=30
```

2.2. Adding an OIDC Provider (e.g., AWS Cognito)

To add Cognito alongside the custom provider:

```
# Enable both providers
auth.provider=custom,cognito

# --- Custom JWT provider (unchanged) ---
quarkus.smallrye-jwt.enabled=true
mp.jwt.verify.publickey.location=publicKey.pem
mp.jwt.verify.issuer=https://myapp.example.com/issuer
mp.jwt.verify.audiences=my-api-client
auth.jwt.secret=${JWT_SECRET}
auth.jwt.expiration=15
auth.jwt.refresh-expiration=30

# --- OIDC / Cognito provider ---
quarkus.oidc.enabled=true
quarkus.oidc.auth-server-url=https://cognito-
```

```
idp.${aws.cognito.region}.amazonaws.com/${aws.cognito.user-pool-id}
quarkus.oidc.client-id=${aws.cognito.clientId}
quarkus.oidc.token.issuer=https://cognito-
idp.${aws.cognito.region}.amazonaws.com/${aws.cognito.user-pool-id}
quarkus.oidc.roles.role-claim-path=cognito:groups

# Cognito environment variables
aws.cognito.user-pool-id=${USER_POOL_ID}
aws.cognito.client-id=${APPLICATION_CLIENT_ID}
aws.cognito.region=us-east-1
aws.cognito.jwks.url=https://cognito-
idp.${aws.cognito.region}.amazonaws.com/${aws.cognito.user-pool-id}/.well-
known/jwks.json
```

2.3. Using Keycloak or Auth0

The pattern is the same — configure `quarkus.oidc.*` to point at the provider's OIDC discovery endpoint:

```
# Keycloak
quarkus.oidc.auth-server-url=https://keycloak.example.com/realms/my-realm
quarkus.oidc.client-id=my-client
quarkus.oidc.credentials.secret=${KEYCLOAK_CLIENT_SECRET}
quarkus.oidc.roles.role-claim-path=realm_access/roles

# Auth0
quarkus.oidc.auth-server-url=https://my-tenant.auth0.com
quarkus.oidc.client-id=my-client-id
quarkus.oidc.credentials.secret=${AUTH0_CLIENT_SECRET}
quarkus.oidc.roles.role-claim-path=https://my-app.example.com/roles
```

Quarkus OIDC automatically discovers the `.well-known/openid-configuration` endpoint from the `auth-server-url`, so you do not need to specify individual endpoints for token validation, JWKS, or user info.

Chapter 3. Issuer Resolution in Detail

When a JWT arrives at the `SecurityFilter`, the following resolution path executes:

```
Incoming request with Authorization: Bearer <token>
    |
    v
SecurityFilter extracts JWT
    |
    v
Parse the "iss" claim from the token
    |
    v
CanonicalIdentityService.validateAccessTokenCanonical(token, issuer)
    |
    v
AuthProviderFactory.getProviderForIssuer(issuer)
    |
    +-- Iterate configured providers:
    |     provider.getIssuer().equals(issuer) ?
    |       -> yes: use this provider
    |       -> no: try next
    |
    +-- No match: fall back to default provider
    |
    v
Provider validates token and returns ProviderClaims or SecurityIdentity
    |
    v
IdentityAssembler merges roles from:
    - TOKEN (from JWT claims)
    - CREDENTIAL (from credential store)
    - USERGROUP (from UserProfile -> UserGroup membership)
    |
    v
Canonical SecurityIdentity with userId as principal
```

Each provider’s <code>getIssuer()</code> method returns its expected issuer string. For the custom JWT provider, this is the value of <code>mp.jwt.verify.issuer</code>. For an OIDC provider, it is typically the <code>auth-server-url</code> (e.g., <code><a href="https://cognito-idp.us-east-1.amazonaws.com/<pool-id&gt" class="bare">https://cognito-idp.us-east-1.amazonaws.com/<pool-id&gt;</code>).

3.1. Role Provenance Tracking

After token validation, the framework tracks the source of every role via `RoleAssignment`:

Source	Meaning
`TOKEN`	Role came from the JWT claims (e.g., Cognito `cognito:groups` or Keycloak `realm_access/roles`)
`CREDENTIAL`	Role stored in the credential record in MongoDB
`USERGROUP`	Role inherited from `UserGroup` membership via the user's `UserProfile`

The final `SecurityIdentity` contains the union of all three sources. Login and token-refresh responses include a `List<RoleAssignment>` so clients and audit logs can trace exactly where each role originated.

Chapter 4. Creating a New Auth Provider

To add a new authentication provider:

1. Create an `@ApplicationScoped` CDI bean that extends `BaseAuthProvider` and implements `AuthProvider` and `UserManagement`.
2. Return a stable name from `getName()` (e.g., `"saml"`, `"apikey"`, `"azure-ad"`).
3. Implement `getIssuer()` to return the issuer string that your tokens will carry.
4. Implement `validateAccessToken(token)` to parse, verify, and build a `SecurityIdentity`.
5. Optionally implement `ClaimsAuthProvider.validateTokenToClaims(token)` for the modern claims-based flow.
6. Add your provider name to `auth.provider` in `application.properties`.

```
@ApplicationScoped
public class MyOidcProvider extends BaseAuthProvider
        implements AuthProvider, ClaimsAuthProvider, UserManagement {

    @ConfigProperty(name = "my.oidc.issuer")
    String issuer;

    @Override
    public String getName() {
        return "my-oidc";
    }

    @Override
    public String getIssuer() {
        return issuer;
    }

    @Override
    public ProviderClaims validateTokenToClaims(String token) {
        // Validate token against the OIDC provider's JWKS
        // Extract subject, roles, attributes
        // Return ProviderClaims
    }

    // ... implement remaining AuthProvider and UserManagement methods
}
```

Then activate it:

```
auth.provider=custom,my-oidc
my.oidc.issuer=https://my-idp.example.com
```

See Authentication and Authorization for the full `AuthProvider` and `UserManagement` interface contracts and `BaseAuthProvider` capabilities.

Chapter 5. Key Pair Management

5.1. Default Keys Module (`quantum-default-keys`)

The framework ships a convenience module, `quantum-default-keys`, that places a default RSA key pair (`privateKey.pem` / `publicKey.pem`) on the classpath. This module is declared as an `<optional>true</optional>` dependency of `quantum-jwt-provider`, so it is available during framework development but does **not** propagate transitively to consuming applications.

To use the default keys in your application during development or testing, add the dependency explicitly:

```
<dependency>
    <groupId>com.end2endlogic</groupId>
    <artifactId>quantum-default-keys</artifactId>
    <version>${quantum.version}</version>
</dependency>
```

In production, either:

- **Remove** the `quantum-default-keys` dependency from your POM and supply your own keys externally, or
- **Override** the key locations via `application.properties` (see below) — the configured paths take precedence over any classpath defaults.

The default keys are published in the open-source framework repository and are intended only for development and testing. Never use the default keys in production. Always generate and deploy your own key pair.

5.2. Configurable Key Locations

By default, `TokenUtils` loads keys from classpath resources named `privateKey.pem` and `publicKey.pem`. You can override these locations in `application.properties` to point at external key files:

```
# Load signing key from the filesystem
quantum.jwt.private-key-location=file:/opt/keys/signing.pem

# Load verification key from the filesystem
quantum.jwt.public-key-location=file:/opt/keys/verify.pem
```

Supported path prefixes:

Prefix	Behavior
`file:`	Load from the filesystem (e.g., `file:/opt/keys/signing.pem`)
`classpath:`	Load from the classpath (e.g., `classpath:mykeys/signing.pem`)
(no prefix)	Treated as a classpath resource name (backward compatible with `privateKey.pem`)

When these properties are set, `TokenUtilsConfigurer` applies them at application startup and invalidates any previously cached keys.

5.2.1. Typical Production Setup

```
# Point at externally managed keys — no quantum-default-keys JAR needed
quantum.jwt.private-key-location=file:/etc/quantum/keys/privateKey.pem
quantum.jwt.public-key-location=file:/etc/quantum/keys/publicKey.pem

# SmallRye JWT validation must also reference the public key
mp.jwt.verify.publickey.location=/etc/quantum/keys/publicKey.pem
```

`TokenUtils` handles signing (private key); SmallRye JWT handles verification (public key via `mp.jwt.verify.publickey.location`). Both must reference the same key pair. When you move to external keys, update both the `quantum.jwt.*` properties and `mp.jwt.verify.publickey.location`.

5.3. Full Per-Tenant Design

For the complete multi-phase design including per-tenant MongoDB key storage, key rotation, JWKS endpoints, and vault integration, see the Per-Tenant Key Pairs Design Document.