Authentication and Authorization

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

Table of Contents

Quantum integrates with Quarkus security while providing a pluggable approach to authentication. The repository includes a JWT provider module to get started quickly and an extension surface to replace or complement it.

Chapter 1. JWT Provider

- Module: quantum-jwt-provider
- Purpose: Validate JWTs on incoming requests, populate the security principal, and surface tenant/org/user claims that feed DomainContext.
- Configuration: Standard Quarkus/MicroProfile JWT properties plus custom claim mappings as needed for DataDomain.

Chapter 2. Pluggable Authentication

You can introduce alternative authentication mechanisms (e.g., API keys, SAML/OIDC front-channel tokens exchanged for back-end JWTs, HMAC signatures) by providing CDI beans that integrate with the security layer and emit the same normalized context consumed by DomainContext/RuleContext.

Typical steps:

1. Implement a request filter or identity provider that validates the token/credential.
2. Map identity and tenant claims into a principal model (tenantId, orgRefName, userId, roles).
3. Ensure BaseResource (and other entry points) can derive DomainContext from that principal.

Chapter 3. Creating an Auth Plugin (using the Custom JWT provider as a reference)

An auth plugin is typically a CDI bean that:

- Extends BaseAuthProvider to inherit user-management helpers and persistence utilities.
- Implements AuthProvider to integrate with request-time authentication flows.
- Implements UserManagement to expose CRUD-style operations for users, passwords, and roles.

A concrete provider should:

- Be annotated as a CDI bean (e.g., @ApplicationScoped).
- Provide a stable getName() identifier (e.g., "custom", "oidc", "apikey").
- Use config properties for secrets, issuers, token durations, and any external identity provider details.
- Build a Quarkus SecurityIdentity with the authenticated principal and roles.

Chapter 4. AuthProvider interface (what a provider must implement)

Core methods:

- SecurityIdentity validateAccessToken(String token)
  - Parse and validate the incoming credential (JWT, API key, signature).
  - Return a SecurityIdentity with principal name and roles. Throw a security exception for invalid tokens.
- String getName()
  - A short identifier for the provider. Persisted alongside credentials and used in logs/metrics.
- LoginResponse login(String userId, String password)
  - Credential-based login. Return a structured response:
  - positiveResponse: includes SecurityIdentity, roles, accessToken, refreshToken, expirationTime, and realm/mongodbUrl if applicable.
  - negativeResponse: includes error codes/reason/message for clients to act on (e.g., password change required).
- LoginResponse refreshTokens(String refreshToken)
  - Validate the refresh token, mint a new access token (and optionally a new refresh token), and return a positive response.

Notes:

- Login flow should check force-change-password or equivalent flags and return a negative response when user interaction is required before issuing tokens.
- validateAccessToken should only accept valid, non-expired tokens and construct SecurityIdentity consistently with role mappings used across the platform.

Chapter 5. UserManagement interface (operations your plugin must support)

Typical responsibilities include:

- User lifecycle
  - String createUser(String userId, String password, Set<String> roles, DomainContext domainContext, [optional] DataDomain)
  - void changePassword(String userId, String oldPassword, String newPassword, Boolean forceChangePassword)
  - boolean removeUserWithUserId(String userId)
  - boolean removeUserWithSubject(String subject)
- Role management
  - void assignRolesForUserId(String userId, Set<String> roles)
  - void assignRolesForSubject(String subject, Set<String> roles)
  - void removeRolesForUserId(String userId, Set<String> roles)
  - void removeRolesForSubject(String subject, Set<String> roles)
  - Set<String> getUserRolesForUserId(String userId)
  - Set<String> getUserRolesForSubject(String subject)
- Lookups and existence checks
  - Optional<String> getSubjectForUserId(String userId)
  - Optional<String> getUserIdForSubject(String subject)
  - boolean userIdExists(String userId)
  - boolean subjectExists(String subject)

Return values and exceptions:

- Throw SecurityException or domain-specific exceptions for invalid states (duplicate users, bad password, unsupported hashing).
- Return Optional for lookups that may not find a result.
- For removals, return boolean to communicate whether a record was deleted.

Chapter 6. Leveraging BaseAuthProvider in your plugin

When you extend BaseAuthProvider, you inherit ready-to-use capabilities that reduce boilerplate:

- Impersonation controls
    - enableImpersonationWithUserId / enableImpersonationWithSubject
    - disableImpersonationWithUserId / disableImpersonationWithSubject
    - These set or clear an impersonation filter script and realm regex that downstream services can honor to act on behalf of another identity under controlled scope.
- Realm override helpers
    - enableRealmOverrideWithUserId / enableRealmOverrideWithSubject
    - disableRealmOverrideWithUserId / disableRealmOverrideWithSubject
    - Useful for multi-realm/tenant scenarios, enabling scoped cross-realm behavior.
- Persistence utilities
    - Built-in use of the credential repository to save, update, and delete credentials.
    - Consistent validation of inputs (non-null checks, non-blank checks).
    - Hashing algorithm guardrails to ensure only supported algorithms are used.

Best practices when deriving:

- Always set the auth provider name in stored credentials so records can be traced to the correct provider.
- Reuse the role merge/remove patterns to avoid accidental role loss.
- Prefer emitting precise exceptions (e.g., NotFound for missing users, SecurityException for access violations).

Chapter 7. Implementing your own provider

Checklist:

- Class design
    - @ApplicationScoped bean
    - extends BaseAuthProvider
    - implements AuthProvider and UserManagement
    - return a stable getName()
- Configuration
    - Externalize secrets (signing keys), issuers, token durations, and realm details via MicroProfile Config.
- SecurityIdentity
    - Consistently build identities with principal and roles; include useful attributes for auditing/telemetry.
- Tokens/credentials
    - For JWT-like tokens, implement robust parsing, signature verification, expiration checks, and claim validation.
    - For non-JWT credentials (API keys, HMAC), ensure replay protection and scope binding.
- Responses and errors
    - Use structured LoginResponse for both success and error paths.
    - Prefer idempotent user/role operations; validate inputs and surface actionable messages.

Chapter 8. CredentialUserIdPassword model and DomainContext

This section explains how user credentials are represented, how those records tie to tenancy and realms, and how the server chooses the database ("realm") for REST calls.

What the credential model represents

userId

The human-friendly login handle that users type. Must be unique within the applicable tenancy/realm scope.

subject

A stable, system-generated identifier for the principal. Tokens and internal references favor subject over userId because subjects do not change.

description, emailOfResponsibleParty

Optional metadata to describe the credential and provide an owner contact.

domainContext

The tenancy and organization placement of the principal. It contains:

- tenantId: Logical tenant partition.
- orgRefName: Organization/business unit within the tenant.
- accountId: Account or billing identifier.
- defaultRealm: The default database/realm used for this identity's operations.
- dataSegment: Optional partitioning segment for advanced sharding or data slicing.

roles

The set of authorities granted (e.g., USER, ADMIN). These become groups/roles on the SecurityIdentity.

issuer

An identifier for who issued the credential or tokens (useful for auditing and multi-provider setups).

passwordHash, hashingAlgorithm

The stored password hash and declared algorithm. Not exposed over REST. Providers verify passwords against this.

forceChangePassword

Flag that forces a password reset on next login; the login flow returns a structured negative response instead of tokens.

lastUpdate

Timestamp for auditing and token invalidation strategies.

area2RealmOverrides

Optional map to route specific functional areas to different realms than the default (e.g., “Reporting” → analytics-realm).

realmRegEx

Optional regex to limit or override which realms this identity may act in; also used by impersonation/override flows.

impersonateFilterScript

Optional script indicating the filter/scope applied during impersonation so actions are constrained.

authProviderName

The name of the provider that owns this credential (e.g., “custom”, “oidc”), enabling multi-provider operations and audits.

How DomainContext selects the realm for REST calls

- For each authenticated request, the server derives or retrieves a DomainContext associated with the principal.
- The DomainContext.defaultRealm indicates which backing MongoDB database (“realm”) should be used by repositories for that request.
- If realm override features are enabled (e.g., through provider helpers or per-credential overrides), the system may route certain functional areas to alternate realms using area2RealmOverrides or validated by realmRegEx.
- The remainder of DomainContext (tenantId, orgRefName, accountId, dataSegment) is applied as scope constraints through permission rules and repository filters so reads and writes are automatically restricted to the correct tenant/org segment.

Typical flow

1. Login
    - A user authenticates with userId/password (or other mechanism).
    - On success, a token is returned alongside role information; the principal is associated with a DomainContext that includes the defaultRealm.
2. Subsequent REST calls
    - The token is validated; the server reconstructs SecurityIdentity and DomainContext.
    - Repositories choose the datastore for defaultRealm and enforce tenant/org filters using the DomainContext values.
    - If the request targets a functional area with a defined override, the operation may route to a different realm for that area alone.
3. UI implications
    - The client does not need to know which realm is selected; it simply calls the API. The server ensures the correct database is used based on DomainContext and any configured overrides.

Best practices

- Keep userId immutable once established; use subject for internal joins and token subjects.
- Always attach the correct DomainContext when creating users to avoid cross-tenant leakage.
- Use realm overrides deliberately for well-isolated areas (e.g., analytics, archiving) and document them for operators.

Chapter 9. Quarkus OIDC out-of-the-box and integrating with common IdPs

Quarkus ships with first-class OpenID Connect (OIDC) support, enabling both service-to-service and browser-based logins.

What the Quarkus OIDC extension provides

- OIDC client and server-side adapters:
  - Authorization Code flow with PKCE for browser sign-in.
  - Bearer token authentication for APIs (validating access tokens on incoming requests).
  - Token propagation for downstream calls (forwarding or exchanging tokens).
- Token verification and claim mapping:
  - Validates issuer, audience, signature, expiration, and scopes.
  - Maps standard claims (sub, email, groups/roles) into the security identity.
- Multi-tenancy and configuration:
  - Supports multiple OIDC tenants via configuration, each with its own issuer, client id/secret, and flows.
- Logout and session support:
  - Front-channel and back-channel logout hooks depending on provider capabilities.

Integrating with common providers

- Works with providers like Keycloak, Auth0, Okta, Azure AD, Cognito, and enterprise IdPs exposing OIDC.
- Configure the issuer URL and client credentials. Quarkus discovers endpoints via the provider’s .well-known/openid-configuration.
- For roles/permissions, map provider groups/roles claims to your platform roles in the identity.

OIDC vs OAuth vs OpenID (terminology and evolution)

OAuth 2.0

- Authorization framework for delegated access (scopes), not authentication. Defines flows to obtain access tokens for APIs.

OpenID (OpenID 1.x/2.0)

- Older federated identity protocol that preceded OIDC. It has been superseded by OpenID Connect.

OpenID Connect (OIDC)

- An identity layer on top of OAuth 2.0. Adds standardized authentication, user info endpoints, ID tokens (JWT) with subject and profile claims, and discovery metadata.
- In practice, OIDC is the modern standard for SSO and user authentication; OAuth remains

the authorization substrate underneath.

Summary

- OpenID → historical, replaced by OIDC.
- OAuth 2.0 → authorization framework.
- OIDC → authentication (identity) layer built on OAuth 2.0.

OIDC and SAML in relation to SSO SAML (Security Assertion Markup Language)::

- XML-based federation protocol widely used in enterprises for browser SSO.
- Uses signed XML assertions transported through browser redirects/posts.

  OIDC
- JSON/REST-oriented, uses JWTs, and is well-suited for modern SPAs and APIs.
- Relationship:
- Both enable SSO and federation across identity providers and service providers.
- Many enterprise IdPs support both; OIDC is generally simpler for APIs and modern web stacks, while SAML is entrenched in legacy/enterprise SSO.
- Bridging:
- Gateways or identity brokers can translate SAML assertions to OIDC tokens and vice versa, allowing gradual migration.

Common customer IdP models and OIDC integration patterns

- Centralized IdP (single-tenant):
  - One organization-wide IdP issues tokens for all users.
  - Configure a single OIDC tenant in Quarkus; map groups/roles to application roles.
- Multi-tenant SaaS with per-tenant IdP:
  - Each customer brings their own IdP (BYOID).
  - Configure Quarkus OIDC multitenancy with per-tenant issuer discovery and client credentials.
  - Tenant selection can be based on domain, request header, or path; the selected OIDC tenant performs login and token validation.
- Brokered identity:
  - Use a broker (e.g., a central identity layer) that federates to multiple upstream IdPs (OIDC, SAML).
  - Quarkus integrates with the broker as a single OIDC client; the broker handles IdP routing and protocol translation.
- Hybrid API and web flows:
  - Browser apps use Authorization Code flow with sessions; APIs use bearer token authentication.
  - Quarkus OIDC extension can handle both in the same application when properly configured.

Best practices

- Prefer OIDC for new integrations; use SAML through a broker if enterprise constraints require it.
- Normalize roles/claims server-side so downstream authorization (RuleContext, repositories) sees consistent group names regardless of IdP.
- Use token exchange or client credentials for service-to-service calls; do not reuse end-user tokens where not appropriate.
- For multi-tenant OIDC, secure tenant resolution logic and validate issuer/tenant binding to prevent mix-ups.

Chapter 10. Authorization via RuleContext

Authentication establishes identity; RuleContext enforces what the identity can do. For each action (CREATE, UPDATE, VIEW, DELETE, ARCHIVE), RuleContext can:

- Allow or deny the action
- Contribute additional filters (e.g., org scoping, functional-area specific sharing)
- Adjust UIActionList to reflect permitted next steps

This division of responsibilities keeps providers focused on identity while policies remain centralized in RuleContext.

Chapter 11. Generating JWT Tokens for API Access

Applications frequently need machine-to-machine access that does not go through an interactive login flow. The framework provides two mechanisms: **short-lived user tokens** generated via the login endpoint, and **long-lived service tokens** generated via the dedicated service-token endpoint. Both produce standard RS256-signed JWTs that carry identity, roles, and scope claims consumed by the `SecurityFilter` and the permission system.

11.1. Token Anatomy

Every JWT produced by the framework contains the following claims:

Claim	Example	Purpose
`sub`	`a1b2c3d4-…` (UUID)	The subject — a stable identifier for the principal. For user tokens, this is the user's `subject` from the credential store. For service tokens, a newly generated UUID.
`iss`	`https://myapp.example.com/issuer`	Issuer — must match `mp.jwt.verify.issuer` in the consuming application's config for the token to be accepted.
`aud`	`b2bi-api-client`	Audience — the intended recipient. Auth tokens use `b2bi-api-client`; refresh tokens use `b2bi-api-client-refresh`.
`groups`	`["admin", "user"]`	Roles embedded in the token. These are merged with roles from the credential store and user group membership at validation time.
`scope`	`authToken`	Distinguishes auth tokens (`authToken`) from refresh tokens (`refreshToken`).
`iat`	`1707849600`	Issued-at timestamp (seconds since epoch).
`exp`	`1707853200`	Expiration timestamp (seconds since epoch).

The token is signed with the application's RSA private key using the RS256 algorithm. By default, the key is loaded from `privateKey.pem` on the classpath (provided by the `quantum-default-keys` module during development). In production, configure `quantum.jwt.private-key-location` to point at an externally managed key. Consumers validate the signature using the corresponding public key (`publicKey.pem` or the configured `quantum.jwt.public-key-location`).

11.2. Generating User Tokens via Login

The standard way to obtain a token is through the login endpoint:

```
curl -sS -X POST \
  'http://localhost:8080/auth/login?userId=admin@example.com&password=secret'
```

Response:

```
{
  "accessToken": "eyJhbGciOiJSUzI1NiIsInR5cCI6...",
  "refreshToken": "eyJhbGciOiJSUzI1NiIsInR5cCI6..."
}
```

The access token expires after the configured duration (`com.b2bi.jwt.duration` in seconds). Use the refresh token to obtain a new access token without re-authenticating:

```
curl -sS -X POST \
  -H 'Authorization: Bearer <refresh-token>' \
  'http://localhost:8080/auth/refresh'
```

This flow is appropriate for interactive clients (UIs, CLI tools) that can handle periodic token refresh.

11.3. Service Tokens for System-to-System Integration

For integrations that cannot perform interactive login — MCP servers, cron jobs, webhook handlers, CI/CD pipelines, partner APIs — the framework provides **service tokens**: long-lived JWTs with a dedicated credential record.

11.3.1. Generating a Service Token

```
curl -sS -X POST \
  -H 'Authorization: Bearer <admin-access-token>' \
  -H 'Content-Type: application/json' \
  http://localhost:8080/auth/service-token \
  -d '{
    "roles": ["user"],
    "expirationSeconds": 31536000,
    "description": "MCP server integration token"
  }'
```

Response:

```
{
  "accessToken": "eyJhbGciOiJSUzI1NiIsInR5cCI6...",
  "issuer": "https://myapp.example.com/issuer",
  "subject": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
  "credentialType": "SERVICE_TOKEN"
}
```

11.3.2. Request Parameters

Field	Type	Required	Description
`roles`	string array	Yes	Roles to embed in the token and store on the credential (e.g., `["user"]`, `["admin"]`, `["agent-read"]`).
`expirationSeconds`	long	No	Seconds until expiry. When `null`, the token is issued for approximately 100 years (effectively non-expiring).
`description`	string	No	Human-readable label for the credential (e.g., `"Nightly sync job"`, `"Claude MCP integration"`).

11.3.3. What Happens on the Backend

When you call `POST /auth/service-token`, the framework:

1. **Validates the caller** — the endpoint requires `admin` or `system` role (`@RolesAllowed({"admin", "system"})`). Only administrators can create service tokens.
2. **Looks up the caller's credential** — from the JWT `sub` claim, the framework finds the caller's `CredentialUserIdPassword` in the system realm.
3. **Creates a new `SERVICE_TOKEN` credential** with:
   - `userId`: `svc:<uuid>` (auto-generated, not user-controlled)
   - `subject`: A new UUID
   - `credentialType`: `SERVICE_TOKEN`
   - `parentCredentialSubject`: The caller's subject (for audit trail)
   - `roles`: From the request
   - `domainContext`: Inherited from the caller's credential (same realm, tenant, org)
   - `issuer`: The configured `mp.jwt.verify.issuer`
4. **Saves the credential** to MongoDB.
5. **Links it to the caller's UserProfile** via `additionalCredentialRefs` so the token can be traced to its creator.
6. **Signs and returns the JWT**.

11.3.4. Using a Service Token

The service token is used like any other JWT — pass it in the `Authorization` header:

```
curl -sS -X GET \
  -H 'Authorization: Bearer <service-token>' \
  http://localhost:8080/api/query/rootTypes
```

The SecurityFilter validates the token, looks up the SERVICE_TOKEN credential by its sub claim, and builds a PrincipalContext with the credential's DomainContext, roles, and realm — exactly as it would for a user token.

11.3.5. Example: Configuring an MCP Server with a Service Token

Generate a token for your MCP integration:

```
# Login as admin first
ACCESS_TOKEN=$(curl -sS -X POST \
  'http://localhost:8080/auth/login?userId=admin@mycompany.com&password=admin-secret'
\
  | jq -r '.accessToken')

# Generate a service token with read-only roles
SERVICE_TOKEN=$(curl -sS -X POST \
  -H "Authorization: Bearer $ACCESS_TOKEN" \
  -H 'Content-Type: application/json' \
  http://localhost:8080/auth/service-token \
  -d '{
    "roles": ["user"],
    "expirationSeconds": 31536000,
    "description": "Claude MCP integration - read-only"
  }' | jq -r '.accessToken')

echo "Service token: $SERVICE_TOKEN"
```

Then configure Claude Desktop to use it via a bridge that includes the token:

```
{
  "mcpServers": {
    "quantum": {
      "command": "node",
      "args": ["/path/to/quantum-mcp-bridge/dist/index.js"],
      "env": {
        "QUANTUM_BASE_URL": "https://api.mycompany.com",
        "QUANTUM_AUTH_TOKEN": "<paste-service-token-here>"
      }
    }
  }
}
```

For the native MCP server (when the Quantum app itself serves /mcp), authentication depends on your deployment. In development, the MCP endpoint may be accessible without a token if you configure it appropriately. In production, configure your MCP client or reverse proxy to include the service token as a Bearer header.

11.4. How Tokens Scope to Security Policies

A token does not grant unlimited access. Every API call passes through the same security pipeline regardless of whether it was made by a human user, an MCP client, or a service integration.

11.4.1. Token-to-Policy Flow

```
JWT arrives in Authorization header
    |
    v
SecurityFilter parses JWT
    |
    v
Extract "sub" claim -> look up CredentialUserIdPassword in system realm
    |
    v
Build PrincipalContext:
    - userId (from credential)
    - roles (merged from TOKEN + CREDENTIAL + USERGROUP)
    - dataDomain (tenantId, orgRefName, accountNum, ownerId)
    - defaultRealm (from credential's DomainContext)
    - area2RealmOverrides (optional per-area realm routing)
    |
    v
Determine ResourceContext:
    - area (from @FunctionalMapping, e.g., "integration")
    - functionalDomain (e.g., "query")
    - action (from @FunctionalAction or HTTP method, e.g., "find")
    |
    v
RuleContext.checkRules(principalContext, resourceContext):
    - Match rules by identity (role or userId)
    - Match by area, functionalDomain, action
    - Apply DataDomain constraints (tenantId, orgRefName, etc.)
    - Return ALLOW / DENY with winning rule name
    |
    v
If DENY -> HTTP 403 with decision details
If ALLOW (EXACT) -> proceed, no data filters
If ALLOW (SCOPED) -> proceed with data-level filters applied at repo layer
```

11.4.2. Role Merging

When a token is validated, the effective roles come from three sources:

1. **TOKEN**: The `groups` claim in the JWT itself.
2. **CREDENTIAL**: The `roles` array stored on the `CredentialUserIdPassword` document in MongoDB.
3. **USERGROUP**: Roles inherited from `UserGroup` membership, resolved via the user’s `UserProfile`

in the target realm.

The union of all three sources becomes the effective role set for the request.

11.4.3. Data Scoping via DataDomain

The credential’s `DomainContext` determines which tenant, organization, and account the token operates within. This translates to a `DataDomain` that is applied as query filters at the repository layer:

- **tenantId**: Isolates data by tenant. A service token created by an `acme-corp` admin can only access `acme-corp` data.
- **orgRefName**: Scopes to an organization within the tenant.
- **accountNum**: Scopes to an account.
- **ownerId**: Set to the credential’s userId; used for owner-based access rules.

These constraints are enforced automatically by Morphia repositories and cannot be bypassed by the token holder.

11.4.4. Realm Override with `X-Realm`

If the service token’s credential has `authorizedRealms` or a matching `realmRegEx`, the caller can pass `X-Realm: <target-realm>` to operate in a different realm. The `SecurityFilter` validates that the credential is authorized for the target realm, then rebuilds the `PrincipalContext` with the target realm’s `DomainContext`.

```
curl -sS -X POST \
  -H 'Authorization: Bearer <service-token>' \
  -H 'X-Realm: acme-corp-staging' \
  -H 'Content-Type: application/json' \
  http://localhost:8080/api/agent/execute \
  -d '{"tool": "query_find", "arguments": {"rootType": "Order", "query":
"status:OPEN"}}'
```

11.4.5. Limiting Service Token Scope

To follow the principle of least privilege:

- **Assign minimal roles** — give the service token only the roles it needs. If it only reads data, do not include `admin`.
- **Use per-tenant agent configuration** — configure `quantum.agent.tenant.<realm>.enabledTools` to restrict which gateway tools the agent can call.
- **Set `maxFindLimit`** — cap query results with `quantum.agent.tenant.<realm>.maxFindLimit`.
- **Use `expirationSeconds`** — even for long-lived tokens, set an expiration. A 1-year token (`31536000` seconds) is preferable to an effectively infinite one.
- **Monitor and audit** — the `parentCredentialSubject` on `SERVICE_TOKEN` credentials links every

service token to the admin who created it. Token validation logs include the userId and realm for each request.

11.5. Generating RSA Key Pairs

The framework signs tokens with an RSA key pair. To generate a new key pair:

```
# Generate a 2048-bit RSA private key in PKCS#8 format
openssl genpkey -algorithm RSA -out privateKey.pem -pkeyopt rsa_keygen_bits:2048

# Extract the public key
openssl rsa -pubout -in privateKey.pem -out publicKey.pem
```

> ❗ The private key must be in PKCS#8 PEM format (begins with `-----BEGIN PRIVATE KEY-----`). PKCS#1 format (`-----BEGIN RSA PRIVATE KEY-----`) is not supported. If you have a PKCS#1 key, convert it:

```
openssl pkcs8 -topk8 -inform PEM -outform PEM -nocrypt -in rsa_private.pem -out
privateKey.pem
```

11.5.1. Where to Place the Keys

There are three options, from simplest to most production-ready:

Option 1: Use the default keys (development only)

Add the `quantum-default-keys` module to your application POM. This places a pre-generated key pair on the classpath automatically — no files to copy.

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

> ⚠ The default keys are published in the open-source repository. Never use them in production.

Option 2: Classpath keys (simple deployments)

Place `privateKey.pem` and `publicKey.pem` in `src/main/resources/` of your application module. The framework loads them from the classpath at startup and caches them in memory.

Option 3: External keys via configuration (recommended for production)

Store the keys outside the application artifact and configure their locations in `application.properties`:

```
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 also needs the public key path
mp.jwt.verify.publickey.location=/etc/quantum/keys/publicKey.pem
```

This approach allows key rotation without rebuilding or redeploying the application. See Key Pair Management for the full configuration reference.

11.6. Configuration Reference

Property	Default	Description
`auth.provider`	(required)	Comma-separated list of provider names (e.g., `custom`, `custom,cognito`). See Multiple Auth Providers.
`mp.jwt.verify.publickey.location`	(required)	Classpath location of the RSA public key PEM file (e.g., `publicKey.pem`).
`mp.jwt.verify.issuer`	(required)	Expected issuer claim in JWTs. Must match the issuer used during token generation.
`mp.jwt.verify.audiences`	(required)	Comma-separated list of accepted audience values (e.g., `b2bi-api-client,b2bi-api-client-refresh`).
`com.b2bi.jwt.duration`	`10000`	Access token duration in seconds for the custom JWT provider.
`auth.jwt.secret`	(required)	Secret for HMAC-based operations. Use an environment variable: `${JWT_SECRET}`.
`auth.jwt.expiration`	`15`	Fallback access token expiration in minutes.
`auth.jwt.refresh-expiration`	`30`	Refresh token expiration in minutes.
`quantum.jwt.private-key-location`	*(unset — defaults to classpath `privateKey.pem`)*	Path to the RSA private key for token signing. Supports `file:`, `classpath:`, or bare classpath resource names. See Key Pair Management.
`quantum.jwt.public-key-location`	*(unset — defaults to classpath `publicKey.pem`)*	Path to the RSA public key for token verification. Same prefix conventions as the private key property. NOTE: also update `mp.jwt.verify.publickey.location` to match.