Authentication and Authorization

Version 1.2.2-SNAPSHOT, 2025-09-13T18:09:10Z

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.