Table of Contents

Chapter 1. REST: Find, Get, List, Save, Update, Delete

Quantum provides consistent REST resources backed by repositories. Extend BaseResource to expose CRUD quickly and consistently.

1.1. Base Concepts

- BaseResource<T, R extends Repo<T>> provides endpoints for:
- find: query by criteria (filters, pagination)
- get: fetch by id or refName
- list: list all within scope with paging
- save: create
- update: modify existing
- delete: delete or soft-delete/archival depending on model
- UIActionList: derive available actions based on current model state.
- DataDomain filtering is applied across all operations to enforce multi-tenancy.

1.2. Example Resource

```
import com.e2eq.framework.rest.resources.BaseResource;
import jakarta.ws.rs.Path;

@Path("/products")
public class ProductResource extends BaseResource<Product, ProductRepo> {
}
```

1.3. Authorization Layers in REST CRUD

Quantum combines static, identity-based checks with dynamic, domain-aware policy evaluation. In practice you will often use both:

1) Hard-coded permissions via annotations

- Use standard Jakarta annotations like @RolesAllowed (or the framework's @RoleAllow if present) on resource classes or methods to declare role-based checks that must pass before executing an endpoint.
- These checks are fast and decisive. They rely on the caller's roles as established by the current SecurityIdentity.

Example:

```
import jakarta.annotation.security.RolesAllowed;

@RolesAllowed({"ADMIN", "CATALOG_EDITOR"})
@Path("/products")
public class ProductResource extends BaseResource<Product, ProductRepo> {
    // Only ADMIN or CATALOG_EDITOR can access all inherited CRUD endpoints
}
```

2) JWT groups and role mapping

- When using the JWT provider, the token's groups/roles claims are mapped into the Quarkus SecurityIdentity (see the Authentication guide).
- Groups in JWT typically become roles on SecurityIdentity; these roles are what @RolesAllowed/@RoleAllow checks evaluate.
- You can augment or transform roles using a SecurityIdentityAugmentor (see RolesAugmentor in the framework) to add derived roles based on claims or external lookups.

3) RuleContext layered authorization (dynamic policies)

- After annotation checks pass, RuleContext evaluates domain-aware permissions. This layer can:
- Enforce DataDomain scoping (tenant/org/owner)
- Allow cross-tenant reads for specific functional areas when policy permits
- Contribute query predicates and projections to repositories
- Think of @RolesAllowed/@RoleAllow as the coarse-grained gate, and RuleContext as the fine-grained, context-sensitive policy engine.

4) Quarkus SecurityIdentity and SecurityFilter

- Quarkus produces a SecurityIdentity for each request containing principal name and roles.
- The framework's SecurityFilter inspects the incoming request (e.g., JWT) and populates/augments the SecurityIdentity and the derived DomainContext used by RuleContext and repositories.
- BaseResource and underlying repos (e.g., MorphiaRepo) consume SecurityIdentity/DomainContext to apply permissions and filters consistently.

For detailed rule-base matching (URL, headers, body predicates, priorities), see the Permissions section.

1.4. Querying

- Use query parameters or a request body (depending on your API convention) to express filters.
- RuleContext contributes tenant-aware filters and projections automatically.
- See Query Language for the full BIAPIQuery syntax, including array filtering with elemMatch and IN-clause enhancements that accept resolver-provided lists.

1.5. Responses and Schemas

- Models are returned with calculated fields (e.g., actionList) when appropriate.
- OpenAPI annotations in your models/resources integrate with MicroProfile OpenAPI for schema docs.

1.6. Error Handling

- Validation errors (e.g., ImportRequiredField, Size) return helpful messages.
- Rule-based denials return appropriate HTTP statuses (403/404) without leaking cross-tenant metadata.

1.6.1. Query Language (ANTLR-based)

The find/list endpoints accept a filter string parsed by an ANTLR grammar (BIAPIQuery.g4). Use the filter query parameter to express predicates; combine them with logical operators and grouping. Sorting and projection are separate query parameters.

- Operators:
- Equals: ':'
- Not equals: ':!'
- Less than/Greater than: ':<' / ':>'
- Less-than-or-equal/Greater-than-or-equal: ':⇐' / ':>='
- Exists (field present): ':~' (no value)
- In list: ':^' followed by [v1,v2,...]
- Boolean literals: true/false
- Null literal: null
- Logical:
- AND: '&&'
- OR: '||'
- NOT: '!!' (applies to a single allowed expression)
- Grouping: parentheses '(' and ')'
- Values by type:
- Strings: unquoted or quoted with "..."; quotes allow spaces and punctuation
- Whole numbers: prefix with '#' (e.g., #10)
- Decimals: prefix with '' (e.g., 19.99)
- Date: yyyy-MM-dd (e.g., 2025-09-10)
- DateTime (ISO-8601): 2025-09-10T12:30:00Z (timezone supported)
- ObjectId (Mongo 24-hex): 5f1e9b9c8a0b0c0d1e2f3a4b

- Reference by ObjectId: @@5f1e9b9c8a0b0c0d1e2f3a4b
- Variables: ${ownerId|principalId|resourceId|action|functionalDomain|pTenantId|pAccountId|rTenantId|rAccountId|realm|area}

1.7. Simple filters (equals)

```
# string equality
name:"Acme Widget"
# whole number
quantity:#10
# decimal number
price:##19.99
# date and datetime
shipDate:2025-09-12
updatedAt:2025-09-12T10:15:00Z
# boolean
active:true
# null checks
description:null
# field exists
lastLogin:~
# object id equality
id:5f1e9b9c8a0b0c0d1e2f3a4b
# variable usage (e.g., tenant scoping)
dataDomain.tenantId:${pTenantId}
```

1.8. Advanced filters: grouping and AND/OR/NOT

```
# Products that are active and (name contains widget OR gizmo), excluding discontinued
active:true && (name:*widget* || name:*gizmo*) && status:!"DISCONTINUED"

# Shipments updated after a date AND (destination NY OR CA)
updatedAt:>=2025-09-01 && (destination:"NY" || destination:"CA")

# NOT example: items where category is not null and not (price < 10)
category:!null && !!(price:<##10)
```

Notes: - Wildcard matching uses '**: name:*widget** (prefix/suffix/contains). '?' matches a single character. - Use parentheses to enforce precedence; otherwise AND/OR follow standard left-to-right with explicit operators.

1.9. IN lists

```
status:^["OPEN","CLOSED","ON_HOLD"]
```

```
ownerId:^["u1","u2","u3"]
referenceId:^[@@5f1e9b9c8a0b0c0d1e2f3a4b, @@6a7b8c9d0e1f2a3b4c5d6e7f]
```

1.10. Sorting

Provide a sort query parameter (comma-separated fields): - '-' prefix = descending, '+' or no prefix = ascending.

Examples:

```
# single field descending
?sort=-createdAt

# multiple fields: createdAt desc, refName asc
?sort=-createdAt,refName
```

1.11. Projections

Limit returned fields with the projection parameter (comma-separated): - '+' prefix = include, '-' prefix = exclude.

Examples:

```
# include only id and refName, exclude heavy fields
?projection=+id,+refName,-auditInfo,-persistentEvents
```

1.12. End-to-end examples

- GET /products/list?skip=0&limit=50&filter=active:true&&name:*widget*&sort=-updatedAt&projection=+id,+name,-auditInfo
- GET /shipments/list?filter=(destination:"NY"||destination:"CA")&&updatedAt:>=2025-09-01&sort=origin

These features integrate with RuleContext and DataDomain: your filter runs within the tenant/org scope derived from the security context; RuleContext may add further predicates or projections automatically.

Chapter 2. CSV Export and Import

These endpoints are inherited by every resource that extends BaseResource. They are mounted under the resource's base path. For example, PolicyResource at /security/permission/policies exposes:

- GET /security/permission/policies/csv
- POST /security/permission/policies/csv
- POST /security/permission/policies/csv/session
- POST /security/permission/policies/csv/session/{sessionId}/commit
- DELETE /security/permission/policies/csv/session/{sessionId}
- GET /security/permission/policies/csv/session/{sessionId}/rows

Authorization and scoping:

- All CSV endpoints are protected by the same @RolesAllowed("user", "admin") checks as other CRUD operations.
- RuleContext filters and DataDomain scoping apply the same way as list/find; exports stream only what the caller may see, and imports are saved under the same permissions.
- In multi-realm deployments, include your X-Realm header as you do for CRUD; underlying repos resolve realm and domain context consistently.

2.1. Export: GET /csv

Produces a streamed CSV download of the current resource collection.

Query parameters and behavior:

fieldSeparator (default ")

Single character used to separate fields. Typical values: `,`, `;`, `\t`.

requestedColumns (default refName)

Comma-separated list of model field names to include, in output order. If omitted, BaseResource defaults to refName. Nested list extraction is supported with the `[0]` notation on a single nested property across all requested columns (e.g., `addresses[0].city`, `addresses[0].zip`). Indices other than `[0]` are rejected. If the nested list has multiple items, multiple rows are emitted per record (one per list element), preserving other column values.

quotingStrategy (default QUOTE_WHERE_ESSENTIAL)

- QUOTE_WHERE_ESSENTIAL: quote only when needed (when a value contains the separator or quoteChar).
- QUOTE_ALL_COLUMNS: quote every column in every row.

quoteChar (default ")

The character used to surround quoted values.

decimalSeparator (default .)

Reserved for decimal formatting. Note: current implementation ignores this value; decimals are rendered using the locale-independent dot.

charsetEncoding (default UTF-8-without-BOM)

One of: `US-ASCII`, `UTF-8-without-BOM`, `UTF-8-with-BOM`, `UTF-16-with-BOM`, `UTF-16BE`, `UTF-16LE`. "with-BOM" values write a Byte Order Mark at the beginning of the file (UTF-8: `EF BB BF`; UTF-16: `FE FF`).

filter (optional)

ANTLR DSL filter applied server-side before streaming (see Query Language section). Reduces rows and can improve performance.

filename (default downloaded.csv)

Suggested download filename returned via Content-Disposition header.

offset (default 0)

Zero-based index of the first record to stream.

length (default 1000, use -1 for all)

Maximum number of records to stream from offset. Use `-1` to stream all (be mindful of client memory/time).

prependHeaderRow (optional boolean, default false)

When true, the first row contains column headers. Requires requestedColumns to be set (the default refName satisfies this requirement).

preferredColumnNames (optional list)

Overrides header names positionally when `prependHeaderRow=true`. The list length must be ≤ requestedColumns; an empty string entry means "use default field name" for that column.

Response:

- 200 OK with Content-Type: text/csv and Content-Disposition: attachment; filename="...".
- On validation/processing errors, the response status is 400/500 and the body contains a single text line describing the problem (e.g., "Incorrect information supplied: ..."). Unrecognized query parameters are rejected with 400.

Examples:

- Export selected fields with header, custom filename and filter

```
curl -H "Authorization: Bearer $JWT" \
     -H "X-Realm: system-com" \

"https://host/api/products/csv?requestedColumns=id,refName,price&prependHeaderRow=true
&filename=products.csv&filter=active:true&sort=+refName"
```

- Export nested list’s first element across columns

```
# emits one row per address entry when more than one is present
curl -H "Authorization: Bearer $JWT" \

"https://host/api/customers/csv?requestedColumns=refName,addresses[0].city,addresses[0
].zip&prependHeaderRow=true"
```

2.2. Import: POST /csv (multipart)

Consumes a CSV file (multipart/form-data) and imports records in batches. The form field name for the file is file.

Query parameters and behavior:

fieldSeparator (default ")

Single character expected between fields.

quotingStrategy (default QUOTE_WHERE_ESSENTIAL)

Same values as export; controls how embedded quotes are recognized.

quoteChar (default ")

The expected quote character in the file.

skipHeaderRow (default true)

When true, the first row is treated as a header and skipped. Mapping is positional, not by header names.

charsetEncoding (default UTF-8-without-BOM)

The file encoding. “with-BOM” variants allow consuming a BOM at the start.

requestedColumns (required)

Comma-separated list of model field names in the same order as the CSV columns. This positional mapping drives parsing and validation. Nested list syntax `[0]` is allowed with the same constraints as export.

Behavior:

- Each row is parsed into a model instance using type-aware processors (ints, longs, decimals, enums, etc.).
- Bean Validation is applied; rows with violations are collected as errors and not saved; valid rows are batched and saved.
- For each saved batch, insert vs update is determined by refName presence in the repository.
- Response entity includes counts (importedCount, failedCount) and per-row results when available.
- Response headers:

- X-Import-Success-Count: number of rows successfully imported.
- X-Import-Failed-Count: number of rows that failed validation or DB write.
- X-Import-Message: summary message.

Example (direct import):

```
curl -X POST \
  -H "Authorization: Bearer $JWT" \
  -H "X-Realm: system-com" \
  -F "file=@policies.csv" \

"https://host/api/security/permission/policies/csv?requestedColumns=refName,principalI
d,description&skipHeaderRow=true&fieldSeparator=,&quoteChar=\"&quotingStrategy=QUOTE_W
HERE_ESSENTIAL&charsetEncoding=UTF-8-without-BOM"
```

2.3. Import with preview sessions

Use a two-step flow to analyze first, then commit only valid rows.

- POST /csv/session (multipart): analyzes the file and creates a session
  - Same parameters as POST /csv (fieldSeparator, quotingStrategy, quoteChar, skipHeaderRow, charsetEncoding, requestedColumns).
  - Returns a preview ImportResult including sessionId, totals (totalRows, validRows, errorRows), and row-level findings. No data is saved yet.
- POST /csv/session/{sessionId}/commit: imports only error-free rows from the analyzed session
  - Returns CommitResult with inserted/updated counts.
  - DELETE /csv/session/{sessionId}: cancels and discards session state (idempotent; always returns 204).
- GET /csv/session/{sessionId}/rows: page through analyzed rows
  - Query params:
  - skip (default 0), limit (default 50)
  - onlyErrors (default false): when true, returns only rows with errors
  - intent (optional): filter rows by intended action: INSERT, UPDATE, or SKIP

Notes and constraints:

- requestedColumns must reference actual model fields. Unknown fields or multiple different nested properties are rejected (only one nested property across requestedColumns is allowed when using [0]).
- Unrecognized query parameters are rejected with HTTP 400 to prevent silent misconfiguration.
- Very large exports should prefer streaming with sensible length settings or server-side filters to reduce memory and time.

- Imports run under the same security rules as POST / (save). Ensure the caller has permission to create/update the target entities in the chosen realm.

Authentication and Authorization

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 3. 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 4. 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 5. 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 6. 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 7. 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 8. 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 9. 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 10. 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.

Chapter 11. 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 (BYOID)
  - Each customer brings their own IdP.
  - 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 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.
  - The OIDC extension can handle both in the same application when properly configured.

Chapter 12. 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.

12.1. Using Ontology Edges in List Endpoints (optional)

When ontology is enabled and edges are materialized, list endpoints can avoid deep joins or multi-collection traversals by rewriting queries based on semantic relationships.

Pattern A: Wrap BSON with ListQueryRewriter

```
Bson base = Filters.and(existingFilters...);
Bson rewritten = rewriter.rewriteForHasEdge(base, tenantId, "placedInOrg", orgRefName
);
collection.find(rewritten).iterator();
```

Pattern B: Constrain Morphia query by IDs

```
Set<String> ids = edgeDao.srcIdsByDst(tenantId, "orderShipsToRegion", region);
if (!ids.isEmpty()) {
  query.filter(dev.morphia.query.filters.Filters.in("_id", ids));
}
```

Notes

- Always scope by tenantId from DomainContext/RuleContext.
- Index edges on (tenantId, p, dst) and (tenantId, src, p) to keep queries fast.
- See Integrating Ontology for more integration options.