Quantum Framework Documentation

Version 1.2.2-SNAPSHOT, 2025-09-13T02:57:15Z

Table of Contents

This documentation provides user guides and tutorials for mid-level Java developers to build SaaS applications with multi-tenancy on the Quantum framework, in a structure similar to Spring's reference documentation. Artifacts are generated as HTML and PDF via Maven.

Chapter 1. Quarkus Core Features and Architecture

Quarkus is a Kubernetes-native Java stack optimized for fast startup, low memory footprint, and developer productivity. Quantum builds on Quarkus to provide multi-tenant persistence, security, and rule-driven authorization.

Key capabilities: - Developer joy: live reload (dev mode), unified config, test-first ergonomics. - Build-time optimizations: aggressive classpath indexing and ahead-of-time processing to reduce reflection and bytecode scanning at runtime. - Container and cloud native: seamless integration with containers, Kubernetes, health checks, metrics, and config. - Extension ecosystem: rich set of extensions for data, security, messaging, observability, and more.

1.1. GraalVM Native and Polyglot

- Native compilation: Quarkus applications can be compiled to native executables using GraalVM (or Mandrel). Benefits include millisecond startup times and drastically reduced RSS memory, ideal for serverless and microservice workloads.
- Constraints: reflection, dynamic proxies, and some dynamic classloading need explicit configuration or substitution (Quarkus largely automates this, see @RegisterForReflection below).
- Polyglot support: GraalVM provides runtimes for multiple languages (e.g., JavaScript, Python via GraalPy, Ruby, R). Applications can embed polyglot code where appropriate using GraalVM's polyglot APIs. Use judiciously to avoid bloating native images and to maintain clear performance boundaries.

1.2. Arc (Quarkus CDI) and Inversion of Control

Arc is Quarkus' CDI implementation, focused on build-time analysis and small runtime overhead. It implements the Inversion of Control pattern: - You declare components (beans) with scopes and qualifiers. - The container instantiates, wires, and manages their lifecycle. - Your code depends on interfaces and qualifiers rather than concrete implementations.

Common scopes and their semantics: - @ApplicationScoped - One contextual instance for the duration of the application. Arc can proxy such beans and apply interceptors. Recommended default for stateless services and repositories. - @Singleton - Single Java instance managed by the container, but not a normal CDI context. It typically doesn't use client proxies; some CDI features (like certain interceptor/proxy behaviors) may differ. Prefer @ApplicationScoped for CDI beans unless you specifically need @Singleton semantics. - @RequestScoped - One instance per incoming HTTP request. Useful for per-request context holders or lightweight state. - @SessionScoped (when web sessions are enabled) - One instance per HTTP session. Use sparingly due to clustering/state implications. - @Dependent (the default if no scope is declared) - No contextual lifecycle; a new instance is created at every injection point, and its lifecycle is bound to the injecting bean. Good for lightweight, stateful helpers.

Recommendations: - Prefer @ApplicationScoped for stateless services, @RequestScoped for per-request concerns, and @Dependent for small, short-lived helpers. - Choose @Singleton only when you explicitly want a single instance without normal CDI contextual behavior.

1.3. Bean Discovery, Qualifiers, and Programmatic Lookups

- Discovery: Quarkus performs build-time bean discovery using classpath indexing. A class becomes a bean when it has a bean-defining annotation (e.g., a scope such as @ApplicationScoped) or is produced via a producer method/field.
- Qualifiers: Use qualifiers (custom annotations annotated with @Qualifier) to disambiguate multiple implementations of the same interface.
- Programmatic selection with Instance<T>:
- Inject Instance<SomeType> to iterate over all beans of a type or to select by qualifier at runtime.
- Useful for plugin architectures where you discover all provider implementations and choose one based on configuration or request context.
- Remember that Instance<T> is lazy; calling get()/iterator() triggers resolution.

1.4. Reflection Configuration and Jandex Indexes

- @RegisterForReflection
- Native images eliminate reflection metadata by default. Annotate classes that must be available to reflection at runtime (e.g., JSON serializers, frameworks performing reflective access).
- Quarkus extensions often auto-register common frameworks. Use this annotation for your own types when needed, especially DTOs or model classes used by reflection-heavy libraries.
- Jandex indexes
- Quarkus builds a Jandex index of your application and dependencies at build time to analyze annotations and discover beans without scanning at runtime.
- This indexing underpins Arc’s fast startup and small footprint by moving classpath analysis to build time.
- When adding third-party libraries that rely on reflection or dynamic discovery, ensure they either provide Jandex indexes or are properly configured for reflection in native mode.

Chapter 2. Guides

Overview: Building SaaS with Quantum

Quantum is a Quarkus-based framework that accelerates building multi-tenant SaaS platforms on MongoDB. It provides:

- Multi-tenancy primitives for tenant creation, isolation, and data sharing
- A domain-first programming model with Functional Areas, Functional Domains, and Actions
- Data security and contextual evaluation via DataDomain, DomainContext, and RuleContext
- Consistent REST resources for find/get/list/save/update/delete operations
- Pluggable authentication with a provided JWT module and extension points

This guide targets mid-level Java developers and follows a structure similar to Spring's reference docs. Use Maven to generate HTML/PDF: see docs module README for commands.

Chapter 3. SaaS and Multi-Tenancy First

SaaS solutions require:

- Onboarding automation: programmatic tenant creation, freemium/trial flows
- Isolation with selective sharing
- Policy-driven access that adapts to user, org, tenant, and action
- Operational efficiency (observability, cost control, upgradeability)

Quantum’s building blocks address these needs out-of-the-box while remaining flexible to fit your architecture.

Multi-Tenancy Models

Quantum supports multiple multi-tenant models for MongoDB deployments:

Chapter 4. One Tenant per Database (in a MongoDB Cluster)

- Each tenant is mapped to a dedicated MongoDB database within a cluster.
- Strong isolation at the database level; operational controls via MongoDB roles.
- Pros: Simplified backup/restore per tenant; reduced risk of data bleed.
- Cons: More databases to manage (indexes, connections), higher operational overhead.

How Quantum helps:

- DataDomain carries tenant identifiers (e.g., tenantId, ownerId, orgRefName) on each model.
- Repositories can resolve connections/DB selection per tenant, enabling routing to the appropriate database.

Chapter 5. Many Tenants in One Database (Shared Database)

- Multiple tenants share a single database and collections.
- Isolation is enforced at the application layer using DataDomain filters.
- Pros: Fewer databases to manage; efficient index utilization and connection pooling.
- Cons: Strict discipline required to enforce filtering and access rules.

How Quantum helps:

- DataDomain is part of every persisted model, enabling programmatic, rule-based filtering.
- RuleContext and DomainContext can be used to inject tenant-aware filters into repositories and resources.
- Cross-tenant sharing can be modeled by specific DataDomain fields and RuleContext logic granting read access across tenants on a per-functional-area basis.

Chapter 6. Freemium and Trial Tenants

- Programmatically create tenants to support self-service onboarding.
- Attach time-bound or capability-bound policies.
- Use scheduled jobs to convert/expire trials.

Quantum patterns:

- Tenant onboarding service creates a DataDomain scope and any default records.
- Policies are encoded in RuleContext checks to allow or restrict actions based on time, plan, or feature flags. = Modeling with Functional Areas, Domains, and Actions

Quantum organizes your system around three core constructs:

- Functional Area: A broad capability area (e.g., Identity, Catalog, Orders, Collaboration).
- Functional Domain: A cohesive sub-area within an area (e.g., in Collaboration: Partners, Shipments, Tasks).
- Actions: The set of operations applicable to a domain (CREATE, UPDATE, VIEW, DELETE, ARCHIVE, plus domain-specific actions).

These constructs allow:

- Fine-grained sharing: Point specific functional areas to shared databases while others remain strictly segmented.
- Policy composition: Apply RuleContext decisions at the level of area/domain/action.

Chapter 7. DataDomain on Models

All persisted models carry DataDomain (tenantId, orgRefName, ownerId, etc.) for rule-based filtering and cross-tenant sharing.

Example model:

```
import dev.morphia.annotations.Entity;
import lombok.Data;
import lombok.EqualsAndHashCode;
import lombok.NoArgsConstructor;
import lombok.experimental.SuperBuilder;
import com.e2eq.framework.model.persistent.base.BaseModel;

@Entity
@Data
@NoArgsConstructor
@SuperBuilder
@EqualsAndHashCode(callSuper = true)
public class Product extends BaseModel {
    private String sku;
    private String name;

    @Override
    public String bmFunctionalArea() { return "Catalog"; }

    @Override
    public String bmFunctionalDomain() { return "Product"; }
}
```

Chapter 8. Persistence Repositories

Define a repository to persist and query your model. With Morphia:

```
import com.e2eq.framework.model.persistent.morphia.MorphiaRepo;

public interface ProductRepo extends MorphiaRepo<Product> {
    // custom queries can be added here
}
```

Chapter 9. Exposing REST Resources

Expose consistent CRUD endpoints by extending BaseResource.

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

@Path("/products")
public class ProductResource extends BaseResource<Product, ProductRepo> {
    // Inherit find, get, list, save, update, delete endpoints
}
```

With this minimal setup, you get standard REST APIs guarded by RuleContext/DataDomain and enriched with UIAction metadata.

Chapter 10. Lombok in Models

Lombok reduces boilerplate in Quantum models and supports inheritance-friendly builders.

Common annotations you will see:

- @Data: Generates getters, setters, toString, equals, and hashCode.
- @NoArgsConstructor: Required by frameworks that need a no-arg constructor (e.g., Jackson, Morphia).
- @EqualsAndHashCode(callSuper = true): Includes superclass fields in equality and hash.
- @SuperBuilder: Provides a builder that cooperates with parent classes (useful for BaseModel subclasses).

Example:

```
@Data
@NoArgsConstructor
@SuperBuilder
@EqualsAndHashCode(callSuper = true)
public class Product extends BaseModel {
  private String sku;
  private String name;
}
```

Notes: - Prefer @SuperBuilder over @Builder when extending BaseModel/UnversionedBaseModel. - Keep equals/hashCode stable for collections and caches; include callSuper when needed.

Chapter 11. Validation with Jakarta Bean Validation

Quantum uses Jakarta Bean Validation to enforce invariants on models at persist time (and optionally at REST boundaries).

Typical annotations:

- @Size(min=3): String/collection length constraints.
- @Valid: Cascade validation to nested objects (e.g., DataDomain on models).
- @NotNull, @Email, @Pattern, etc., as needed.

Where validation runs:

- Repository layer via Morphia ValidationInterceptor (prePersist):
- Executes validator.validate(entity) before the document is written.
- If there are violations and the entity does not implement InvalidSavable with canSaveInvalid=true, an E2eqValidationException is thrown.
- If DataDomain is null and SecurityContext has a principal, ValidationInterceptor will default the DataDomain from the principal context.
- Optionally at REST boundaries: You may also annotate resource DTOs/parameters with Jakarta validation; Quarkus can validate them before the method executes.

Chapter 12. Jackson vs Jakarta Validation Annotations

These two families of annotations serve different purposes and complement each other:

- Jackson annotations (com.fasterxml.jackson.annotation.*) control JSON serialization/deserialization.
- Examples: @JsonIgnore, @JsonIgnoreProperties, @JsonProperty, @JsonInclude.
- They do not enforce business constraints; they affect how JSON is produced/consumed.
- Jakarta Validation annotations (jakarta.validation.*) declare constraints that are evaluated at runtime.
- Examples: @NotNull, @Size, @Valid, @Pattern.

Correspondence and interplay:

- Use Jackson to hide or rename fields in API responses/requests (e.g., @JsonIgnore on transient/calculated fields such as UIActionList).
- Use Jakarta Validation to ensure incoming/outgoing models satisfy required constraints; ValidationInterceptor runs before persistence to enforce them.
- It’s common to annotate the same field with both families when you both constrain values and want specific JSON behavior.

Chapter 13. Jackson ObjectMapper in Quarkus and in Quantum

How Quarkus creates ObjectMapper:

- Quarkus produces a CDI-managed ObjectMapper. You can customize it by providing a bean that implements io.quarkus.jackson.ObjectMapperCustomizer.
- You can also tweak common features via application.properties using quarkus.jackson.* properties.

Quantum defaults:

- The framework provides a QuarkusJacksonCustomizer that:
- Sets DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES = true (reject unknown JSON fields).
- Registers custom serializers/deserializers for org.bson.types.ObjectId so it can be used as String in APIs.

Snippet from the framework:

```
@Singleton
public class QuarkusJacksonCustomizer implements ObjectMapperCustomizer {
  @Override
  public void customize(ObjectMapper objectMapper) {
    objectMapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, true);
    SimpleModule module = new SimpleModule();
    module.addSerializer(ObjectId.class, new ObjectIdJsonSerializer());
    module.addDeserializer(ObjectId.class, new ObjectIdJsonDeserializer());
    objectMapper.registerModule(module);
  }
}
```

Customize in your app:

- Add another ObjectMapperCustomizer bean (order is not guaranteed; make changes idempotent):

```
@Singleton
public class MyJacksonCustomizer implements ObjectMapperCustomizer {
  @Override
  public void customize(ObjectMapper mapper) {
    mapper.findAndRegisterModules();
    mapper.disable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);
    mapper.setSerializationInclusion(JsonInclude.Include.NON_NULL);
  }
```

```
}
```

- Or set properties in application.properties:

```
# Fail if extraneous fields are present
quarkus.jackson.fail-on-unknown-properties=true
# Example date format and inclusion
quarkus.jackson.write-dates-as-timestamps=false
quarkus.jackson.serialization-inclusion=NON_NULL
```

When to adjust:

- Relax fail-on-unknown only for backward-compatibility scenarios; strictness helps catch client mistakes.
- Register modules (JavaTime, etc.) if your models include those types.

Chapter 14. Validation Lifecycle and Morphia Interceptors

Morphia interceptors enhance and enforce behavior during persistence. Quantum registers the following for each realm-specific datastore:

Order of registration (see MorphiaDataStore): 1) ValidationInterceptor 2) PermissionRuleInterceptor 3) AuditInterceptor 4) ReferenceInterceptor 5) PersistenceAuditEventInterceptor

High-level responsibilities:

- ValidationInterceptor (prePersist):
- Defaults DataDomain from SecurityContext if missing.
- Runs bean validation and throws E2eqValidationException on violations unless the entity supports saving invalid states (InvalidSavable).
- PermissionRuleInterceptor (prePersist):
- Evaluates RuleContext with PrincipalContext and ResourceContext from SecurityContext.
- Throws SecurityCheckException if the rule decision is not ALLOW (enforcing write permissions for save/update/delete).
- AuditInterceptor (prePersist):
- Sets AuditInfo on creation and updates lastUpdate fields on modification; captures impersonation details if present.
- ReferenceInterceptor (prePersist):
- For @Reference fields annotated with @TrackReferences, maintains back-references on the parent entities via ReferenceEntry and persists the parent when needed.
- PersistenceAuditEventInterceptor (prePersist when @AuditPersistence is present):
- Appends a PersistentEvent with type PERSIST, date, userId, and version to the model's persistentEvents before saving.

When does validation occur?

- On every save/update path that hits persistence, prePersist triggers validation (and permission/audit/reference processing) before the document is written to MongoDB, guaranteeing constraints and policies are enforced consistently across all repositories.

Chapter 15. Functional Area/Domain in RuleContext Permission Language

Models express their placement in the business model via: - bmFunctionalArea(): returns a broad capability area (e.g., Catalog, Collaboration, Identity) - bmFunctionalDomain(): returns the specific domain within that area (e.g., Product, Shipment, Partner)

How these map into authorization and rules:

- ResourceContext/DomainContext: When a request operates on a model, the framework derives the functional area and domain from the model type (or resource) and places them on the current context alongside the action (CREATE, UPDATE, VIEW, DELETE, ARCHIVE). RuleContext consumes these to evaluate policies.
- Permission language (headers): Rule bases can match on HTTP headers such as x-functional-area and x-functional-domain. These are often set by the resource layer or middleware to reflect the model's bmFunctionalArea/bmFunctionalDomain for the current operation.
- Permission language (query variables): The ANTLR-based query language exposes variables that can be referenced in filters:
- ${area} corresponds to bmFunctionalArea()
- ${functionalDomain} corresponds to bmFunctionalDomain() These can be used to author reusable filters or to record audit decisions by area/domain.
- Repository filters: RuleContext can contribute additional predicates that are area/domain-specific, enabling fine-grained sharing. For example, a shared Catalog area may allow cross-tenant VIEW, while a Collaboration.Shipment domain remains tenant-strict.

Examples

1) Header-based rule matching (Permissions)

```
- name: allow-catalog-reads
  priority: 300
  match:
    method: [GET]
    url: /api/**
    headers:
      x-functional-area: [Catalog]
      x-functional-domain: [Product, Category]
    rolesAny: [USER, ADMIN]
  effect: ALLOW
  filters:
    readScope: { orgRefName: PUBLIC }
```

2) Query variable usage (Filters)

You can reference the active area/domain in filter expressions (e.g., for auditing or conditional

branching in custom rule evaluators):

```
# Constrain reads differently when operating in the Catalog area
(${area}:"Catalog" && dataDomain.orgRefName:"PUBLIC") ||
(${area}:!"Catalog" && dataDomain.tenantId:${pTenantId})
```

3) Model-driven mapping

Given a model like:

```
@Override public String bmFunctionalArea()  { return "Collaboration"; }
@Override public String bmFunctionalDomain(){ return "Shipment"; }
```

- Incoming REST requests that operate on Shipment resources set area=Collaboration and functionalDomain=Shipment in the ResourceContext.
- RuleContext evaluates policies considering action + area + domain, e.g., deny cross-tenant UPDATE in Collaboration.Shipment, but allow cross-tenant VIEW in Collaboration.Partner if marked shared.

Notes

- If you create composite resources that span multiple models, set the headers (x-functional-area, x-functional-domain) explicitly for each endpoint so rules can target them precisely.
- See also: the Permissions section for rule-base matching and priorities, and the DomainContext/RuleContext section for end-to-end flow.

Chapter 16. StateGraphs on Models

StateGraphs let you restrict valid values and transitions of String state fields. They are declared on model fields with @StateGraph and enforced during save/update when the model class is annotated with @Stateful.

Key pieces: - @StateGraph(graphName="..."): mark a String field as governed by a named state graph. - @Stateful: mark the entity type as participating in state validation. - StateGraphManager: runtime registry that holds graphs and validates transitions. - StringState and StateNode: define the graph (states, initial/final flags, transitions).

Defining a state graph at startup:

```
@Startup
@ApplicationScoped
public class StateGraphInitializer {
  @Inject StateGraphManager stateGraphManager;
  @PostConstruct void init() {
    StringState order = new StringState();
    order.setFieldName("orderStringState");

    Map<String, StateNode> states = new HashMap<>();
    states.put("PENDING",    StateNode.builder().state("PENDING").initialState(true)
.finalState(false).build());
    states.put("PROCESSING", StateNode.builder().state("PROCESSING").initialState
(false).finalState(false).build());
    states.put("SHIPPED",    StateNode.builder().state("SHIPPED").initialState(false)
.finalState(false).build());
    states.put("DELIVERED",  StateNode.builder().state("DELIVERED").initialState(
false).finalState(true).build());
    states.put("CANCELLED",  StateNode.builder().state("CANCELLED").initialState(
false).finalState(true).build());
    order.setStates(states);

    Map<String, List<StateNode>> transitions = new HashMap<>();
    transitions.put("PENDING",    List.of(states.get("PROCESSING"), states.get(
"CANCELLED")));
    transitions.put("PROCESSING", List.of(states.get("SHIPPED"), states.get("
CANCELLED")));
    transitions.put("SHIPPED",    List.of(states.get("DELIVERED"), states.get(
"CANCELLED")));
    transitions.put("DELIVERED",  null);
    transitions.put("CANCELLED",  null);
    order.setTransitions(transitions);

    stateGraphManager.defineStateGraph(order);
  }
}
```

Using the graph in a model:

```
@Stateful
@Entity
@EqualsAndHashCode(callSuper = true)
public class Order extends BaseModel {
  @StateGraph(graphName = "orderStringState")
  private String status;

  @Override public String bmFunctionalArea()  { return "Orders"; }
  @Override public String bmFunctionalDomain(){ return "Order"; }
}
```

How it affects save/update: - On create: validateInitialStates ensures the field value is one of the configured initial states. Otherwise, InvalidStateTransitionException is thrown. - On update: validateStateTransitions checks each @StateGraph field's old→new transition against the graph via StateGraphManager.validateTransition(). If invalid, save/update fails with InvalidStateTransitionException. This applies to full-entity saves and to partial updates via repo.update(...pairs) on that field. - Utilities: StateGraphManager.getNextPossibleStates(graphName, current) and printStateGraph(...) can aid UIs.

Chapter 17. References and EntityReference

Morphia @Reference establishes relationships between entities: - One-to-one: a BaseModel field annotated with @Reference. - One-to-many: a Collection<BaseModel> field annotated with @Reference.

Example:

```
@Entity
public class Shipment extends BaseModel {
  @Reference(ignoreMissing = false)
  @TrackReferences
  private Partner partner;   // parent entity
}
```

EntityReference is a lightweight reference object used across the framework to avoid DBRef loading when only identity info is needed. Any model can produce one:

```
EntityReference ref = shipment.createEntityReference();
// contains: entityId, entityType, entityRefName, entityDisplayName (and optional
realm)
```

REST convenience: - BaseResource exposes GET /entityref to list EntityReference for a model with optional filter/sort. - Repositories expose getEntityReferenceListByQuery(...), and utilities exist to convert lists of EntityReference back to entities when needed.

When to use which: - Use @Reference for strong persistence-level links where Morphia should maintain foreign references. - Use EntityReference for UI lists, foreign-key-like pointers in other documents, events/audit logs, or cross-module decoupling without DBRef behavior.

Chapter 18. Tracking References with @TrackReferences and Delete Semantics

@TrackReferences on a @Reference field tells the framework to maintain a back-reference set on the parent entity. The back-reference field is UnversionedBaseModel.references (a Set<ReferenceEntry>), which is calculated/maintained by the framework and should not be set by clients.

What references contains: - Each ReferenceEntry holds: referencedId (ObjectId of the child), type (fully-qualified class name of the child's entity), and refName (child's stable reference name). - It indicates that the parent is being referenced by the given child entity. The set is used for fast checks and to enforce referential integrity.

How tracking works (save/update): - ReferenceInterceptor inspects @Reference fields annotated with @TrackReferences during prePersist. - When a child references a parent, a ReferenceEntry for the child is added to the parent's references set and the parent is saved to persist the back-reference. - For @Reference collections, entries are added for each child-parent pair. - If a @Reference is null but ignoreMissing=false, a save will fail with an IllegalStateException since the parent is required.

How it affects delete: - During delete in MorphiaRepo.delete(...): - If obj.references is empty, the object can be deleted directly (after removing any references it holds to parents). - If obj.references is not empty, the repo checks each ReferenceEntry. If any referring parent still exists, a ReferentialIntegrityViolationException is thrown to prevent breaking relationships. - If all references are stale (referring objects no longer exist), the repo removes stale entries, removes this object's own reference constraints from parents, and performs the delete within a transaction. - removeReferenceConstraint(...) ensures that, when deleting a child, its ReferenceEntry is removed from parent.references and the parent is saved, keeping back-references consistent.

Practical guidance: - Annotate parent links with both @Reference and @TrackReferences when you need strong integrity guarantees and easy "who references me?" queries. - Use ignoreMissing=true only for optional references; you still get back-reference tracking when not null. - Expect HTTP delete to fail with a meaningful error if there are live references; remove or update those references first, or design cascading behavior explicitly in your domain logic.

DomainContext, RuleContext, and DataDomain

Quantum enforces multi-tenant isolation and sharing through contextual data carried on models and evaluated at runtime.

Chapter 19. DataDomain

Every persisted model includes a DataDomain that describes ownership and scope, commonly including fields such as:

- tenantId: Identifies the tenant
- orgRefName: Organization unit reference within a tenant
- ownerId: Owning user or system entity
- realm: Optional runtime override for partitioning

These fields enable filtering, authorization, and controlled sharing of data between tenants or org units.

Chapter 20. DomainContext

DomainContext represents the current execution context for a request or operation, typically capturing:

- current tenant/org/user identity
- functional area / functional domain
- the action being executed (e.g., CREATE, UPDATE, VIEW, DELETE, ARCHIVE)

It feeds downstream components (repositories, resources) to consistently apply filtering and policy decisions.

Chapter 21. RuleContext

RuleContext encapsulates policy evaluation. It can:

- Enforce whether an action is allowed for a given model and DataDomain
- Produce additional filters and projections used by repositories
- Grant cross-tenant read access for specific functional areas (e.g., shared catalogs) while keeping others strictly isolated

Chapter 22. End-to-End Flow

1. A REST request enters a BaseResource-derived endpoint.
2. The resource builds a DomainContext from the security principal and request parameters.
3. RuleContext evaluates permissions and returns effective filters.
4. Repository applies filters (DataDomain-aware) to find/get/list/update/delete.
5. The model's UIActionList can be computed to reflect what the caller can do next.

This pattern ensures consistent enforcement across all CRUD operations, independent of the specific model or repository.

REST: Find, Get, List, Save, Update, Delete

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

Chapter 23. 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.

Chapter 24. Example Resource

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

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

Chapter 25. 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.

Chapter 26. Querying

- Use query parameters or a request body (depending on your API convention) to express filters.
- RuleContext contributes tenant-aware filters and projections automatically.

Chapter 27. 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.

Chapter 28. 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.

Chapter 29. 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}

29.1. 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}
```

29.2. 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.

29.3. IN lists

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

29.4. 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
```

29.5. 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
```

29.6. 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 30. 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.

30.1. 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"
```

30.2. 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"
```

30.3. 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.

Database Migrations and Index Management

This guide explains Quantum's MongoDB migration subsystem (quantum-morphia-repos), how migrations are authored and executed, and how to manage indexes. It also documents the REST APIs that trigger migrations and index operations.

Chapter 31. Overview

Quantum uses a simple, versioned change-set mechanism to evolve MongoDB schemas and seed data safely across realms (databases). Key building blocks:

- ChangeSetBean: a CDI bean describing one migration step with metadata (from/to version, priority, etc.) and an execute method.
- ChangeSetBase: convenience base class you can extend; provides logging helpers and optional targeting controls.
- MigrationService: discovers pending change sets, applies them in order within a transaction, records execution, and bumps the DatabaseVersion.
- DatabaseVersion and ChangeSetRecord: stored in Mongo to track current schema version and previously executed change sets.

Chapter 32. Semantic Versioning

Semantic Versioning (SemVer) expresses versions in the form MAJOR.MINOR.PATCH (for example, 1.4.2):

- MAJOR: increment for incompatible/breaking schema changes.
- MINOR: increment for backward-compatible additions (new collections/fields that don't break existing code).
- PATCH: increment for backward-compatible fixes or small adjustments.

Why this matters for migrations: - Ordering: migrations must apply in a deterministic order that reflects real compatibility. SemVer provides a natural ordering and clear intent for authors and reviewers. - Compatibility checks: the application can assert that the current database is "new enough" to run the code safely.

How semver4j is used: - Parsing and validation: version strings are parsed into a SemVer object. Invalid strings fail fast during parsing, ensuring only compliant versions are stored and compared. - Introspection and comparison: the parsed object exposes major/minor/patch components and supports comparisons, enabling safe ordering and "greater than / less than" checks. - Consistent string form: the canonical string is retained for display, logs, and API responses.

How DatabaseVersion leverages SemVer: - Single source of truth: DatabaseVersion stores the canonical SemVer string alongside a parsed SemVer object for logic and comparisons. - Efficient ordering: for fast sorting and tie-breaking, DatabaseVersion also keeps a compact integer encoding of MAJOR.MINOR.PATCH as (major*100) + (minor*10) + patch (e.g., 1.0.3 → 103). This makes numeric comparisons straightforward while still recording the exact SemVer string. - Migration flow: when migrations run, successful execution records the new database version in DatabaseVersion. Startup checks compare the stored version to the required quantum.database.version to prevent the app from running against an older, incompatible schema.

Recommendations: - Always bump MAJOR for breaking data changes, MINOR for additive changes, and PATCH for backward-compatible fixes. - Keep change sets small and target a single to-version per change set to make intent clear. - Use SemVer consistently in getDbFromVersion/getDbToVersion across all change sets so ordering and compatibility checks remain reliable.

Chapter 33. Configuration

The following MicroProfile config properties influence migrations:

- quantum.database.version: target version the application requires (SemVer, e.g., 1.0.3). MigrationService.checkDataBaseVersion compares this to the stored version.
- quantum.database.migration.enabled: feature flag checked by resources/services when running migrations. Default: true.
- quantum.database.migration.changeset.package: package containing change sets (CDI still discovers beans via type, but this property documents the intended package).
- quantum.realmConfig.systemRealm, quantum.realmConfig.defaultRealm, quantum.realmConfig.testRealm: well-known realms used by MigrationResource when running migrations across environments.

Chapter 34. How change sets are discovered and executed

- Discovery: MigrationService#getAllChangeSetBeans locates all CDI beans implementing ChangeSetBean.
- Ordering: change sets are sorted by dbToVersionInt, then by priority (ascending). That ensures lower target versions apply before higher ones; priority resolves ties.
- Pending selection: For the target realm, MigrationService#getAllPendingChangeSetBeans compares each change set's dbToVersion against the stored DatabaseVersion and ignores already executed ones (tracked in ChangeSetRecord).
- Locking: A distributed lock (Mongo-backed Sherlock) is acquired per realm before applying change sets to prevent concurrent execution.
- Transactions: Each change set runs within a MorphiaSession transaction; on success the change is recorded in ChangeSetRecord and DatabaseVersion is advanced (if higher). On failure the transaction is aborted and the error returned.
- Realms: Migrations run per realm (Mongo database). A change set can optionally be restricted to certain database names or even override the realm it executes against (see below).

Chapter 35. Authoring a change set

Implement ChangeSetBean; most change sets extend ChangeSetBase.

Required metadata methods:

- getId(): a string id for human tracking (e.g., 00003)
- getDbFromVersion() / getDbFromVersionInt(): previous version you are migrating from (SemVer and an int like 102 for 1.0.2)
- getDbToVersion() / getDbToVersionInt(): target version after running this change (SemVer and int)
- getPriority(): integer priority when multiple change sets have same toVersion
- getAuthor(), getName(), getDescription(), getScope(): informational fields recorded in ChangeSetRecord

Execution method:

- void execute(MorphiaSession session, MongoClient mongoClient, MultiEmitter<? super String> emitter)
- Perform your data/index changes using the provided session (transaction).
- Use emitter.emit("message") to stream log lines back to SSE clients.

Optional targeting controls (provided by ChangeSetBase):

- boolean isOverrideDatabase(): return true to execute against a specific database instead of the requested realm.
- String getOverrideDatabaseName(): the concrete database name to use when overriding.
- Set<String> getApplicableDatabases(): return a set of database names to which this change set should apply. Return null or an empty set to allow all.

Logging helper:

- ChangeSetBase.log(String, MultiEmitter) emits to both Quarkus log and the SSE stream.

Chapter 36. Example change sets in the framework

Package: com.e2eq.framework.model.persistent.morphia.changesets

- InitializeDatabase
- Seeds foundational data in a new realm: counters (e.g., accountNumber), system Organization and Account, initial Rule and Policy scaffolding, default user profiles and security model. Uses EnvConfigUtils and SecurityUtils to derive system DataDomain and defaults.
- AddAnonymousSecurityRules
- Adds a defaultAnonymousPolicy with an allow rule for unauthenticated actions such as registration and contact-us in the website area.
- AddRealms
- Creates the system and default Realm records based on configuration, if missing.

These are typical examples of idempotent change sets that can be safely re-evaluated.

Chapter 37. REST APIs to trigger migrations (MigrationResource)

Base path: /system/migration

Security: Most endpoints require admin role; dbversion is PermitAll for introspection.

- GET /system/migration/dbversion/{realm}
- Returns the current DatabaseVersion document for the given realm, or 404 if not found.
- Example: curl -s http://localhost:8080/system/migration/dbversion/system-com
- POST /system/migration/indexes/applyIndexes/{realm}
- Admin only. Calls MigrationService.applyIndexes(realm) which invokes Morphia Datastore.applyIndexes() for all mapped entities. Use this after adding @Indexed annotations.
- Example: curl -X POST -H "Authorization: Bearer $TOKEN" http://localhost:8080/system/migration/indexes/applyIndexes/system-com
- POST /system/migration/indexes/dropAllIndexes/{realm}
- Admin only. Drops all indexes on all mapped collections in the realm. Useful before re-creation or when changing index definitions.
- Example: curl -X POST -H "Authorization: Bearer $TOKEN" http://localhost:8080/system/migration/indexes/dropAllIndexes/system-com
- POST /system/migration/initialize/{realm}
- Admin only. Server-Sent Events (SSE) stream that executes all pending change sets for the specific realm.
- Example (note -N to keep connection open): curl -N -X POST -H "Authorization: Bearer $TOKEN" http://localhost:8080/system/migration/initialize/system-com
- GET /system/migration/start
- Admin only. SSE stream that runs pending change sets across test, system, and default realms from configuration.
- Example: curl -N -H "Authorization: Bearer $TOKEN" http://localhost:8080/system/migration/start
- GET /system/migration/start/{realm}
- Admin only. SSE for a specific realm.
- Example: curl -N -H "Authorization: Bearer $TOKEN" http://localhost:8080/system/migration/start/my-realm

SSE responses stream human-readable messages produced by MigrationService and your change sets. The connection ends with "Task completed" or an error message.

Chapter 38. Per-entity index management (BaseResource)

Every entity resource that extends BaseResource<T, R extends BaseMorphiaRepo<T>> exposes a convenience endpoint to (re)create indexes for a single collection in a realm.

- POST <entity-resource-base-path>/indexes/ensureIndexes/{realm}?collectionName=<collection>
- Admin only. Invokes R.ensureIndexes(realm, collectionName).
- Use this when you want to reapply indexes for one collection without touching others.
- Example (assuming a ProductResource at /products): curl -X POST -H "Authorization: Bearer $TOKEN" \ "http://localhost:8080/products/indexes/ensureIndexes/system-com?collectionName=product"

Chapter 39. Global index management (MigrationService)

MigrationService also exposes programmatic index utilities used by the MigrationResource endpoints:

- applyIndexes(realm): calls Morphia Datastore.applyIndexes() for the realm.
- dropAllIndexes(realm): iterates mapped entities and drops indexes on each underlying collection.

Chapter 40. Validating versions at startup

- MigrationService.checkDataBaseVersion() compares the stored DatabaseVersion in each well-known realm to quantum.database.version and throws a DatabaseMigrationException when lower than required. This prevents the app from running against an incompatible schema.
- MigrationService.checkInitialized(realm) is a convenience that asserts DatabaseVersion exists and is >= required version; helpful for preflight checks.

Chapter 41. Notes and best practices

- Make change sets idempotent: Always check for existing records before creating/updating indexes or documents.
- Use SemVer consistently for from/to versions. The framework computes an integer form (e.g., 1.0.3 → 103) for ordering.
- Prefer small, focused change sets with clear descriptions and authorship.
- Use the MultiEmitter in execute(…) to provide progress to operators consuming the SSE endpoint.
- Apply new indexes with applyIndexes after deploying models with new @Indexed annotations; optionally dropAllIndexes then applyIndexes when changing index definitions across the board.
- Limit scope: use getApplicableDatabases() to constrain execution to specific databases, or isOverrideDatabase/getOverrideDatabaseName to target a different database when appropriate. = 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 42. 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 43. 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 44. 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 45. 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 46. 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 47. 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 48. 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 49. 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 50. 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 51. 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.

Permissions: Rule Bases, SecurityURLHeaders, and SecurityURLBody

This section explains how Quantum evaluates permissions for REST requests using rule bases that match on URL, HTTP method, headers, and request body content. It also covers how identities and roles (as found on userProfile or credentialUserIdPassword) are matched, how priority works, and how multiple matching rule bases are evaluated.

Note: The terms SecurityURLHeaders and SecurityURLBody in this document describe the matching dimensions for rules. Implementations may vary, but the semantics below are stable for authoring and reasoning about permissions.

Chapter 52. Key Concepts

- Identity: The authenticated principal, typically originating from JWT or another provider. It includes:
- userId (or credentialUserIdPassword username)
- roles (authorities/groups)
- tenantId, orgRefName, optional realm, and other claims that contribute to DomainContext
- userProfile: A domain representation of the user that aggregates identity, roles, and policy decorations (feature flags, plans, expiration, etc.).
- Rule Base (Permission Rule): A declarative rule with matching criteria and an effect (ALLOW or DENY). Criteria may include:
- HTTP method and URL pattern
- SecurityURLHeaders: predicates over selected HTTP headers (e.g., x-functional-area, x-functional-domain, x-tenant-id)
- SecurityURLBody: predicates over request body fields (JSON paths) or query parameters
- Required roles/attributes on identity or userProfile
- Functional area/domain/action
- Priority: integer used to sort rule evaluation
- Effect: ALLOW or DENY; an ALLOW may also contribute scope filters (e.g., DataDomain constraints) to be applied downstream by repositories.

Chapter 53. Rule Structure (Illustrative)

```
- name: allow-public-reads
  priority: 100
  match:
    method: [GET]
    url: /api/catalog/**
    headers:
      x-functional-area: [Catalog]
    rolesAny: [USER, ADMIN]
  effect: ALLOW
  filters:
    # Optional: contribute additional DataDomain filters
    readScope: { orgRefName: PUBLIC }

- name: deny-non-admin-delete
  priority: 10
  match:
    method: [DELETE]
    url: /api/**
  requireRolesAll: [ADMIN]
  effect: ALLOW

- name: default-deny
  priority: 10000
  match: {}
  effect: DENY
```

- headers under match are the SecurityURLHeaders predicates.
- Body predicates (SecurityURLBody) can be expressed similarly as JSONPath-like constraints:

```
  body:
    $.dataDomain.tenantId: ${identity.tenantId}
    $.action: [CREATE, UPDATE]
```

Chapter 54. Matching Algorithm

Given a request R and identity I, evaluate a set of rule bases RB as follows:

1. Candidate selection
    - From RB, select all rules whose URL pattern and HTTP method match R.
2. Attribute and header/body checks
    - For each candidate, check:
    - SecurityURLHeaders: header predicates must all match (case-insensitive header names; values support exact string, regex, or one-of lists depending on rule authoring capability).
    - SecurityURLBody: if present, evaluate body predicates against parsed JSON body (or query params when body is absent). Predicates must all match.
    - Identity/UserProfile: role requirements and attribute requirements must be satisfied.
3. Priority sort
    - Sort matching candidates by ascending priority (lower numbers indicate higher precedence). If not specified, default priority is 1000.
4. Evaluation order and decision
    - Iterate in sorted order; the first rule that yields a decisive effect (ALLOW or DENY) becomes the decision.
    - If the rule is ALLOW and contributes filters (e.g., DataDomain read/write scope), attach those to the request context for downstream repositories.
5. Multi-match aggregation (optional advanced mode)
    - In advanced configurations, if multiple ALLOW rules match at the same priority, their filters may be merged (intersection for restrictive scope, union for permissive scope) according to a configured merge strategy. If not configured, the default is first-match-wins.
6. Fallback
    - If no rules match decisively, apply a default policy (typically DENY).

Chapter 55. Priorities

- Lower integer = higher priority. Example: priority 1 overrides priority 10.
- Use tight scopes with low priority for critical protections (e.g., denies), and broader ALLOWs with higher numeric priority.
- Recommended ranges:
- 1–99: global deny rules and emergency blocks
- 100–499: domain/area-specific critical rules
- 500–999: standard ALLOW policies
- 1000+: defaults and catch-alls

Chapter 56. Multiple Matching RuleBases

- First-match-wins (default): after sorting by priority, the first decisive rule determines the result; subsequent matches are ignored.
- Merge strategy (optional):
- When enabled and multiple ALLOW rules share the same priority, scopes/filters are merged.
- Conflicts between ALLOW and DENY at the same priority resolve to DENY unless explicitly configured otherwise (fail-safe).

Chapter 57. Identity and Role Matching

- RolesAny: request is allowed if identity has at least one of the specified roles.
- RolesAll: request requires all listed roles.
- Attribute predicates can compare identity/userProfile attributes (e.g., identity.tenantId == header.x-tenant-id).
- Time or plan-based conditions: userProfile can embed plan and expiration; rules may check that trials are active or features are enabled.

Chapter 58. Example Scenarios

1) Public catalog browsing - Request: GET /api/catalog/products?search=widgets - Headers: x-functional-area=Catalog - Identity: anonymous or role USER - Rules: - allow-public-reads (priority 100) ALLOW + readScope orgRefName=PUBLIC - Outcome: ALLOW; repository applies DataDomain filter orgRefName=PUBLIC

2) Tenant-scoped shipment update - Request: PUT /api/shipments/ABC123 - Headers: x-functional-area=Collaboration, x-tenant-id=T1 - Body: { dataDomain: { tenantId: "T1" }, … } - Identity: user in tenant T1 with roles [USER] - Rules: - allow-collab-update (priority 300) requires body.dataDomain.tenantId == identity.tenantId and rolesAny USER, ADMIN ⇒ ALLOW - Outcome: ALLOW; Rule contributes writeScope tenantId=T1

3) Cross-tenant admin read with higher priority - Request: GET /api/partners - Identity: role ADMIN (super-admin) - Rules: - admin-override (priority 50) ALLOW - default-tenant-read (priority 600) ALLOW with tenant filter - Outcome: admin-override wins due to higher precedence (lower number), allowing broader read

4) Conflicting ALLOW and DENY at same priority - Two rules match with priority 200: one ALLOW, one DENY - Resolution: DENY wins unless merge strategy configured to handle explicitly; recommended to avoid same-priority conflicts by policy.

Chapter 59. Operational Tips

- Author specific DENY rules with low numbers to prevent accidental exposure.
- Keep URL patterns narrowly tailored for sensitive domains.
- Prefer header/body predicates to refine matches without exploding URL patterns.
- Log matched rule names and applied scopes for auditability.

Chapter 60. How UIActions and DefaultUIActions are calculated

When the server returns a collection of entities (for example, userProfiles), each entity may expose two action lists: - DefaultUIActions: the full set of actions that conceptually apply to this type of entity (e.g., CREATE, UPDATE, VIEW, DELETE, ARCHIVE). Think of this as the “menu template” for the type. - UIActions: the subset of actions the current user is actually permitted to perform on that specific entity instance right now.

Why they can differ per entity: - Entity attributes: state or flags (e.g., archived, soft-deleted, immutable) can remove or alter available actions at instance level. - Permission rule base: evaluated against the current request, identity, and context to allow or deny actions. - DataDomain membership: tenant/org/owner scoping can further restrict actions if the identity is outside the entity’s domain.

How the server computes them: 1) Start with a default action template for the entity type (DefaultUIActions). 2) Apply simple state-based adjustments (e.g., suppress CREATE on already-persisted instances). 3) Evaluate the permission rules with the current identity and context: - Consider roles, functional area/domain, action intent, headers/body, and any rule-contributed scopes. - Resolve DataDomain constraints to ensure the identity is permitted to act within the entity’s domain. 4) Produce UIActions as the allowed subset for that entity instance. 5) Return both lists with each entity in collection responses.

How the client should use the two lists: - Render the full DefaultUIActions as the visible set of possible actions (icons, buttons, menus) so the UI stays consistent. - Enable only those actions present in UIActions; gray out or disable the remainder to signal capability but lack of current permission. - This approach avoids flicker and keeps affordances discoverable while remaining truthful to the user’s current authorization.

Example: - You fetch 25 userProfiles. - DefaultUIActions for the type = [CREATE, VIEW, UPDATE, DELETE, ARCHIVE]. - For a specific profile A (owned by your tenant), UIActions may be [VIEW, UPDATE] based on your roles and domain. - For another profile B (in a different tenant), UIActions may be [VIEW] only. - The UI renders the same controls for both A and B, but only enables the actions present in each item’s UIActions list.

Operational considerations: - Keep action names stable and documented so front-ends can map to icons and tooltips consistently. - Prefer small, composable rules that evaluate action permissions explicitly by functional area/domain to avoid surprises. - Consider server-side caching of action evaluations for list views to reduce latency, respecting identity and scope.

Chapter 61. How This Integrates End-to-End

- BaseResource extracts identity and headers to construct DomainContext.
- Rule evaluation uses URL/method + SecurityURLHeaders + SecurityURLBody + identity/userProfile to reach a decision and derive scope filters.
- Repositories (e.g., MorphiaRepo) apply the filters to queries and updates, ensuring DataDomain-respecting access.

Chapter 62. Administering Policies via REST (PolicyResource)

The PolicyResource exposes CRUD-style REST APIs for creating and managing policies (rule bases) that drive authorization decisions. Each Policy targets a principalId (either a specific userId or a role name) and contains an ordered list of Rule objects. Rules match requests using SecurityURIHeader and SecurityURIBody and then contribute an effect (ALLOW/DENY) and optional repository filters.

- Base path: /security/permission/policies
- Auth: Bearer JWT (see Authentication); resource methods are guarded by @RolesAllowed("user", "admin") at the BaseResource level and your own realm/role policies.
- Multi-realm: pass X-Realm header to operate within a specific realm; otherwise the default realm is used.

62.1. Model shape (Policy)

A Policy extends FullBaseModel and includes: - id, refName, displayName, dataDomain, archived/expired flags (inherited) - principalId: userId or role name that this policy attaches to - description: human-readable summary - rules: array of Rule entries

Rule fields (key ones): - name, description - securityURI.header: identity, area, functionalDomain, action (supports wildcard "") - **securityURI.body: realm, orgRefName, accountNumber, tenantId, ownerId, dataSegment, resourceId (supports wildcard** "") - effect: ALLOW or DENY - priority: integer; lower numbers evaluated first - finalRule: boolean; stop evaluating when this rule applies - andFilterString / orFilterString: ANTLR filter DSL snippets injected into repository queries (see Query Language section)

Example payload:

```
{
  "refName": "defaultUserPolicy",
  "displayName": "Default user policy",
  "principalId": "user",
  "description": "Users can act on their own data; deny dangerous ops in security
area",
  "rules": [
    {
      "name": "view-own-resources",
      "description": "Limit reads to owner and default data segment",
      "securityURI": {
        "header": { "identity": "user", "area": "*", "functionalDomain": "*", "action
": "*" },
        "body":   { "realm": "*", "orgRefName": "*", "accountNumber": "*", "tenantId":
"*", "ownerId": "*", "dataSegment": "*", "resourceId": "*" }
      },
```

```
      "andFilterString": "
dataDomain.ownerId:${principalId}&&dataDomain.dataSegment:#0",
      "effect": "ALLOW",
      "priority": 300,
      "finalRule": false
    },
    {
      "name": "deny-delete-in-security",
      "securityURI": {
        "header": { "identity": "user", "area": "security", "functionalDomain": "*",
"action": "delete" },
        "body":   { "realm": "*", "orgRefName": "*", "accountNumber": "*", "tenantId":
"*", "ownerId": "*", "dataSegment": "*", "resourceId": "*" }
      },
      "effect": "DENY",
      "priority": 100,
      "finalRule": true
    }
  ]
}
```

62.2. Endpoints

All endpoints are relative to /security/permission/policies. These are inherited from BaseResource and are consistent across entity resources.

- GET /list
- Query params: skip, limit, filter, sort, projection
- Returns a Collection<Policy> with paging metadata; respects X-Realm.
- GET /id/{id} and GET /id?id=...
- Fetch a single Policy by id.
- GET /refName/{refName} and GET /refName?refName=...
- Fetch a single Policy by refName.
- GET /count?filter=...
- Returns a CounterResponse with total matching entities.
- GET /schema
- Returns JSON Schema for Policy.
- POST /
- Create or upsert a Policy (if id is present and matches an existing entity in the selected realm, it is updated).
- PUT /set?id=...&pairs=field:value
- Targeted field updates by id. pairs is a repeated query parameter specifying field/value pairs.

- PUT /bulk/setByQuery?filter=…&pairs=…
- Bulk updates by query. Note: ignoreRules=true is not supported on this endpoint.
- PUT /bulk/setByIds
- Bulk updates by list of ids posted in the request body.
- PUT /bulk/setByRefAndDomain
- Bulk updates by a list of (refName, dataDomain) pairs in the request body.
- DELETE /id/{id} (or /id?id=…)
- Delete by id.
- DELETE /refName/{refName} (or /refName?refName=…)
- Delete by refName.
- CSV import/export endpoints for bulk operations:
- GET /csv – export as CSV (field selection, encoding, etc.)
- POST /csv – import CSV into Policies
- POST /csv/session – analyze CSV and create an import session (preview)
- POST /csv/session/{sessionId}/commit – commit a previously analyzed session
- DELETE /csv/session/{sessionId} – cancel a session
- GET /csv/session/{sessionId}/rows – page through analyzed rows
- Index management (admin only):
- POST /indexes/ensureIndexes/{realm}?collectionName=policy

Headers: - Authorization: Bearer <token> - X-Realm: realm identifier (optional but recommended in multi-tenant deployments)

Filtering and sorting: - filter uses the ANTLR-based DSL (see REST CRUD > Query Language) - sort uses comma-separated fields with optional +/- prefix; projection accepts a comma-separated field list

62.3. Examples

- Create or update a Policy

```
curl -X POST \
  -H "Authorization: Bearer $JWT" \
  -H "Content-Type: application/json" \
  -H "X-Realm: system-com" \
  https://host/api/security/permission/policies \
  -d @policy.json
```

- List policies for principalId=user

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

"https://host/api/security/permission/policies/list?filter=principalId:'user'&sort=+re
fName&limit=50"
```

- Delete a policy by refName

```
curl -X DELETE \
  -H "Authorization: Bearer $JWT" \
  -H "X-Realm: system-com" \
  "https://host/api/security/permission/policies/refName/defaultUserPolicy"
```

62.4. How changes affect rule bases and enforcement

- Persistence vs. in-memory rules:
- PolicyResource updates the persistent store of policies (one policy per principalId or role with a list of rules).
- RuleContext is the in-memory evaluator used by repositories and resources to enforce permissions. It matches SecurityURIHeader/Body, orders rules by priority, and applies effects and filters.
- Making persisted policy changes effective:
- On startup, migrations (see InitializeDatabase and AddAnonymousSecurityRules) typically seed default policies and/or programmatically add rules to RuleContext.
- When you modify policies via REST, you have two options to apply them at runtime: 1) Implement a reload step that reads policies from PolicyRepo and rehydrates RuleContext (e.g., RuleContext.clear(); then add rules built from current policies). 2) Restart the service or trigger whatever policy-loader your application uses at boot.
- Tip: If you maintain a background watcher or admin endpoint to refresh policies, keep it tenant/realm-aware and idempotent.
- Evaluation semantics (recap):
- Rules are sorted by ascending priority; the first decisive rule sets the outcome. finalRule=true stops further processing.
- andFilterString/orFilterString contribute repository filters through RuleContext.getFilters(), constraining result sets and write scopes.
- principalId can be a concrete userId or a role; RuleContext considers both the principal and all associated roles.
- Safe rollout:
- Create new policies with a higher numeric priority (lower precedence) first, test with GET /schema and dry-run queries.
- Use realm scoping via X-Realm to stage changes in a non-production realm.

- Prefer DENY with low priority numbers for critical protections.

See also: - Permissions: Matching Algorithm, Priorities, and Multiple Matching RuleBases (sections above) - REST CRUD: Query Language and generic endpoint behaviors

Chapter 63. Tutorials

Tutorial: Supply Chain Collaboration End-to-End

In this tutorial, we will model a simplified supply chain collaboration system, create repositories and REST resources, and see how Quantum provides a robust set of APIs that a UI can utilize—all with multi-tenancy baked in.

Chapter 64. Domain Overview

Functional Area: Collaboration

Functional Domains:

- Partner: Companies participating in the supply chain (manufacturers, carriers, retailers)
- Shipment: Units of transport moving goods between locations
- Task: Collaborative tasks associated with shipments (e.g., confirm pickup, update ETA)

Chapter 65. Models

```
import dev.morphia.annotations.Entity;
import lombok.Data;
import lombok.EqualsAndHashCode;
import lombok.NoArgsConstructor;
import lombok.experimental.SuperBuilder;
import com.e2eq.framework.model.persistent.base.BaseModel;

@Entity
@Data
@NoArgsConstructor
@SuperBuilder
@EqualsAndHashCode(callSuper = true)
public class Partner extends BaseModel {
    private String refCode;
    private String name;
    @Override public String bmFunctionalArea() { return "Collaboration"; }
    @Override public String bmFunctionalDomain() { return "Partner"; }
}

@Entity
@Data
@NoArgsConstructor
@SuperBuilder
@EqualsAndHashCode(callSuper = true)
public class Shipment extends BaseModel {
    private String trackingNumber;
    private String origin;
    private String destination;
    @Override public String bmFunctionalArea() { return "Collaboration"; }
    @Override public String bmFunctionalDomain() { return "Shipment"; }
}
```

DataDomain on BaseModel ensures tenant context (tenantId/orgRefName/etc.) is carried and indexed as needed (see idx_refIdUniqueInDomain).

Chapter 66. Repositories

```
import com.e2eq.framework.model.persistent.morphia.MorphiaRepo;

public interface PartnerRepo extends MorphiaRepo<Partner> {}
public interface ShipmentRepo extends MorphiaRepo<Shipment> {}
```

Chapter 67. REST Resources

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

@Path("/partners")
public class PartnerResource extends BaseResource<Partner, PartnerRepo> {}

@Path("/shipments")
public class ShipmentResource extends BaseResource<Shipment, ShipmentRepo> {}
```

These resources automatically expose consistent find/get/list/save/update/delete endpoints with DataDomain-aware filtering.

Chapter 68. Multi-Tenant Sharing Scenarios

- Shared Partner Directory: Allow cross-tenant read of Partner records while keeping Shipments strictly tenant-scoped. Implement in RuleContext by permitting VIEW on Partner across tenants when a sharing flag is set in DataDomain (e.g., orgRefName == "PUBLIC").
- Strict Shipment Isolation: Enforce that Shipment queries always constrain by tenantId and (optionally) orgRefName.

Chapter 69. RuleContext Sketch

```
public class CollaborationRuleContext /* implements your RuleContext SPI */ {
    public boolean canView(Object model, DomainContext ctx) {
        // Allow partner directory reads across tenants if shared
        if (model instanceof Partner p) {
            return p.getDataDomain() != null && "PUBLIC".equals(p.getDataDomain()
.getOrgRefName());
        }
        // Default: require same tenant
        return ctx.getTenantId().equals(extractTenant(model));
    }
}
```

Chapter 70. Authentication

Use quantum-jwt-provider to accept JWTs from your IdP. Map claims to tenantId/orgRefName/user/roles; BaseResource builds DomainContext for RuleContext.

Chapter 71. Separating Models and Repos for Workflows

Keep models and repositories in their own modules (as this repository does for quantum-models and quantum-morphia-repos). This allows:

- Integration into workflows like Temporal (activities operating directly on repos/models)
- Orchestration by tools like Windmill, while REST APIs remain in another service/module

Chapter 72. Result

With these building blocks, a UI can:

- Search the Partner directory (cross-tenant where allowed)
- Create/manage Shipments scoped to the tenant
- Update and track tasks with action-driven UIActionList feedback

All while maintaining consistent policy enforcement via RuleContext and DataDomain.