

Quantum Tutorial

Build a Multi-tenant SaaS with Quantum

Version 1.2.2-SNAPSHOT, 2025-10-15T01:34:17Z

Table of Contents

1. 1. Project setup	2
1.1. Getting Started: Your First Quantum Application	2
1.1.1. Prerequisites	2
1.1.2. Project Setup	2
1.1.3. Configuration	2
1.1.4. Your First Model	3
1.1.5. Repository	4
1.1.6. REST Resource	4
1.1.7. Running the Application	5
1.1.8. Testing Your API	5
1.1.9. What You Get Automatically	5
1.1.10. Next Steps	6
2. 2. Platform overview	7
2.1. Overview: Building SaaS with Quantum	7
2.1.1. SaaS and Multi-Tenancy First	7
3. 3. Domain modeling fundamentals	8
3.1. Modeling with Functional Areas, Domains, and Actions	8
3.1.1. Prefer Annotations for Functional Mapping (recommended)	8
3.1.2. DataDomain on Models	10
3.1.3. Persistence Repositories	11
3.1.4. Exposing REST Resources	11
3.1.5. Lombok in Models	11
3.1.6. Validation with Jakarta Bean Validation	12
3.1.7. Jackson vs Jakarta Validation Annotations	12
4. Jackson ObjectMapper in Quarkus and in Quantum	14
5. Validation Lifecycle and Morphia Interceptors	16
6. Functional Area/Domain in RuleContext Permission Language	17
7. StateGraphs on Models	19
8. CompletionTasks and CompletionTaskGroups	21
9. References and EntityReference	24
10. Tracking References with @TrackReferences and Delete Semantics	25
11. 4. Multi-tenant model and realm separation	27
11.1. Multi-Tenancy Models	27
11.1.1. One Tenant per Database (in a MongoDB Cluster)	27
11.1.2. Many Tenants in One Database (Shared Database)	27
11.1.3. Freemium and Trial Tenants	28
12. 5. Domain rule context	29
DomainContext, RuleContext, and DataDomain	30

13. DataDomain	31
14. DomainContext	32
15. RuleContext	33
16. End-to-End Flow	34
17. Resolvers and Variables in Rule Filters	35
DomainContext, RuleContext, and DataDomain	36
18. DataDomain	37
19. DomainContext	38
20. RuleContext	39
21. End-to-End Flow	40
22. Resolvers and Variables in Rule Filters	41
23. Concrete example: building and using a resolver	42
23.1. 1) Implement the SPI	42
23.2. 2) How RuleContext uses resolvers	43
23.3. 3) Author a rule that consumes the variable	43
23.4. 4) End-to-end behavior	43
23.5. String literals vs. typed values in resolver variables	43
23.5.1. Example A: Resolver returns ObjectIds (typed)	44
23.5.2. Example B: Resolver returns String literals (force raw strings)	44
23.5.3. Other types supported	45
24. 6. Building RESTful CRUD APIs	46
24.1. REST: Find, Get, List, Save, Update, Delete	46
24.1.1. Base Concepts	46
24.1.2. Example Resource	46
24.1.3. Authorization Layers in REST CRUD	46
24.1.4. Querying	48
24.1.5. Responses and Schemas	48
24.1.6. Error Handling	48
24.1.7. Query Language (ANTLR-based)	48
24.2. Simple filters (equals)	49
24.3. Advanced filters: grouping and AND/OR/NOT	49
24.4. IN lists	50
24.5. Sorting	50
24.6. Projections	50
24.7. End-to-end examples	50
25. CSV Export and Import	51
25.1. Export: GET /csv	51
25.2. Import: POST /csv (multipart)	53
25.3. Import with preview sessions	54
Authentication and Authorization	56
26. JWT Provider	57

27. Pluggable Authentication	58
28. Creating an Auth Plugin (using the Custom JWT provider as a reference)	59
29. AuthProvider interface (what a provider must implement)	60
30. UserManagement interface (operations your plugin must support)	61
31. Leveraging BaseAuthProvider in your plugin	62
32. Implementing your own provider	63
33. CredentialUserIdPassword model and DomainContext	64
34. Quarkus OIDC out-of-the-box and integrating with common IdPs	66
35. Authorization via RuleContext	68
36. 7. Query language and filtering	69
36.1. Query Language Reference	69
36.1.1. Basic Syntax	69
36.1.2. Common Patterns	70
36.1.3. Advanced Examples	72
36.1.4. Variables in Filters	73
36.1.5. Performance Tips	73
36.1.6. Integration with REST APIs	74
36.1.7. Error Handling	75
36.1.8. See Also	75
37. 8. Authentication, permissions, and annotations	76
Authentication and Authorization	77
38. JWT Provider	78
39. Pluggable Authentication	79
40. Creating an Auth Plugin (using the Custom JWT provider as a reference)	80
41. AuthProvider interface (what a provider must implement)	81
42. UserManagement interface (operations your plugin must support)	82
43. Leveraging BaseAuthProvider in your plugin	83
44. Implementing your own provider	84
45. CredentialUserIdPassword model and DomainContext	85
46. Quarkus OIDC out-of-the-box and integrating with common IdPs	88
47. Authorization via RuleContext	91
Permissions: Rule Bases, SecurityURLHeaders, and SecurityURLBody	92
48. Introduction: Layered Enforcement Overview	93
49. Key Concepts	95
50. Rule Structure (Illustrative)	96
51. Matching Algorithm	97
52. Priorities	98
53. Grant-based vs Deny-based Rule Sets	99
54. Feature Flags, Variants, and Target Rules	100
55. Multiple Matching RuleBases	103
56. Identity and Role Matching	104

56.1. How roles are defined for an identity (role sources and resolution)	104
57. Example Scenarios	106
58. Operational Tips	107
59. How UIActions and DefaultUIActions are calculated	108
60. How This Integrates End-to-End	110
61. Administering Policies via REST (PolicyResource)	111
61.1. Model shape (Policy)	111
61.2. Endpoints	112
61.3. Examples	113
61.4. How changes affect rule bases and enforcement	114
62. Realm override (X-Realm) and Impersonation (X-Impersonate)	116
62.1. What they do (at a glance)	116
62.2. How the headers integrate with permission evaluation	116
62.3. Required credential configuration (CredentialUserIdPassword)	117
62.4. End-to-end behavior from SecurityFilter (reference)	117
62.5. Practical differences and use cases	118
62.6. Examples	118
63. Data domain assignment on create: DomainContext and DataDomainPolicy	120
63.1. The problem this solves (and why it matters)	120
63.2. Key concepts recap: DomainContext and DataDomain	120
63.3. The default policy (do nothing and it works)	120
63.4. Policy scopes: principal-attached vs. global	121
63.5. The policy map and matching	121
63.6. How the resolver works	122
63.7. When would you want a non-global policy?	122
63.8. Relation to tenancy models	123
63.9. Authoring tips	123
63.10. API pointers	123
63.10.1. @FunctionalMapping	123
63.10.2. @FunctionalAction	124
63.10.3. Default Action Mapping	124
63.10.4. How the Framework Uses These Annotations	125
63.10.5. Migration from Legacy Methods	125
63.10.6. Best Practices	126
63.10.7. Integration with Permission Rules	126
63.10.8. See Also	127
64. 9. Seed packs: Declarative tenant seeding	128
Seed packs and declarative tenant seeding	129
65. Introduction	130
65.1. The problem	130
65.2. Why this needs to be solved	130

65.3. How seed packs solve it	130
66. Why seed packs?	131
67. High-level flow	132
68. Manifest quick reference	133
69. Programmatic usage	134
70. Extensibility hooks	135
71. Operational tips	136
72. Primary scenarios	137
73. Explicit examples	138
73.1. Example 1: Minimal manifest and NDJSON	138
73.2. Example 2: Applying packs in code	138
73.3. Example 3: Using an archetype	139
73.4. Example 4: Exact version and includes in a manifest	139
74. Troubleshooting	140
75. How seeds are applied automatically at startup	141
76. Transforms in depth	142
76.1. Creating your own transforms (example: DropIfTransform)	144
77. Test walkthrough: SeedLoaderIntegrationTest	147
78. Archetypes explained	148
78.1. Example A: Define and apply an archetype in the same pack	148
78.2. Example B: Cross-pack archetype in a dedicated "editions" pack	149
78.3. Resolution and ordering details	149
78.4. Interaction with ApplySeedPacksChangeSet	149
78.5. Tenant provisioning with archetypes	150
79. REST API for seed packs	151
80. 10. Migrations	153
Database Migrations and Index Management	154
81. Overview	155
82. Semantic Versioning	156
83. Configuration	157
84. How change sets are discovered and executed	158
85. Authoring a change set	159
86. Example change sets in the framework	160
87. REST APIs to trigger migrations (MigrationResource)	161
88. Per-entity index management (BaseResource)	162
89. Global index management (MigrationService)	163
90. Validating versions at startup	164
91. Notes and best practices	165
92. 11. Testing	166
93. Testing in Quantum: Security Contexts, Repos, and REST APIs	167
93.1. Testing Framework	167

93.2. Prerequisites and glossary	167
93.3. Pattern 1 — Extend BaseRepoTest	167
93.4. Pattern 2 — Try-with-resources SecuritySession	168
93.5. Pattern 3 — Scoped-call wrapper (ScopedCallScope)	168
93.6. Pattern 4 — @TestSecurity annotation (Quarkus)	169
93.7. Testing REST APIs vs. Repository Logic	170
93.8. Useful Quarkus Test features	171
93.9. Real-world tips	171
93.10. Summary	171
94. 12. Next steps	172

This tutorial takes you end-to-end through building a small SaaS application on the Quantum framework. It pulls in the core concepts and how-to content from the main index in a progressive, hands-on flow so you can learn by doing.

Audience: Mid-level Java developers building multi-tenant SaaS on Quarkus and MongoDB.

Prerequisites: JDK 17+, Maven, Docker (for local MongoDB), and basic familiarity with Quarkus.

Artifacts: The docs can be built as HTML/PDF via Maven; the code snippets compile against the Quantum modules published in this repo.

What you'll build: A small, tenant-aware service with domain models, REST CRUD, flexible queries, security, migrations, and seed packs for baseline data.

Each section starts with: - Problem: What we are solving - Why for SaaS: Why this matters in multi-tenant systems - How Quantum helps: Concepts and features that address the problem - Walkthrough: Code and configuration that build on prior sections

Throughout, we reuse and expand the same example to reinforce learning. The reference guide links back to these sections for deeper context.

Chapter 1. 1. Project setup

Problem: Getting a runnable project scaffolded with Quantum and Quarkus.

Why for SaaS: Consistent project structure and dependencies are critical to scale teams and environments.

How Quantum helps: Provides opinionated dependencies, conventions, and ready-made modules for multi-tenancy, security, persistence, and seeding.

Walkthrough: Follow the steps below to create and run your first service.

1.1. Getting Started: Your First Quantum Application

This section walks through creating a simple multi-tenant Product catalog to demonstrate core Quantum concepts.

1.1.1. Prerequisites

- Java 17+
- Maven 3.8+
- MongoDB (local or cloud)
- Basic Quarkus knowledge (see [Quarkus Foundation](#))

1.1.2. Project Setup

Create a new Quarkus project with Quantum dependencies:

```
mvn io.quarkus:quarkus-maven-plugin:create \
  -DprojectGroupId=com.example \
  -DprojectArtifactId=product-catalog \
  -DclassName="com.example.ProductResource" \
  -Dpath="/products"
```

Add Quantum dependencies to `pom.xml`:

```
<dependency>
  <groupId>com.e2eq.framework</groupId>
  <artifactId>quantum-framework</artifactId>
  <version>${quantum.version}</version>
</dependency>
```

1.1.3. Configuration

Create `.env` file (copy from template):

```
MONGODB_USERNAME=your-username
MONGODB_PASSWORD=your-password
MONGODB_DATABASE=product-catalog
MONGODB_HOST=localhost:27017
JWT_SECRET=your-secret-key
```

Basic `application.properties`:

```
# MongoDB
quarkus.mongodb.connection-
string=${MONGODB_CONNECTION_STRING:mongodb://localhost:27017}
quarkus.mongodb.database=${MONGODB_DATABASE:product-catalog}

# JWT Authentication
auth.provider=custom
auth.jwt.secret=${JWT_SECRET:change-me-in-production}
auth.jwt.expiration=60

# CORS for development
quarkus.http.cors=true
quarkus.http.cors.origins=http://localhost:3000
```

1.1.4. Your First Model

Create a Product model with multi-tenancy built-in:

```
package com.example.model;

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

import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;
import java.math.BigDecimal;

@Entity
@Data
@NoArgsConstructor
@SuperBuilder
@EqualsAndHashCode(callSuper = true)
@FunctionalMapping(area = "catalog", domain = "product")
public class Product extends BaseModel {
```

```

@NotBlank
@Size(max = 50)
private String sku;

@NotBlank
@Size(max = 200)
private String name;

private String description;
private BigDecimal price;
private boolean active = true;
}

```

Key points: - Extends `BaseModel` for automatic `DataDomain`, audit fields, and ID management - `@FunctionalMapping` declares this model's business area and domain for security rules - Standard Jakarta validation annotations - Lombok reduces boilerplate

1.1.5. Repository

Create a repository interface:

```

package com.example.repository;

import com.e2eq.framework.model.persistent.morphia.MorphiaRepo;
import com.example.model.Product;

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

```

1.1.6. REST Resource

Create a REST endpoint:

```

package com.example.resource;

import com.e2eq.framework.rest.resources.BaseResource;
import com.example.model.Product;
import com.example.repository.ProductRepo;
import jakarta.ws.rs.Path;

@Path("/products")
public class ProductResource extends BaseResource<Product, ProductRepo> {
    // Inherits all CRUD endpoints: GET, POST, PUT, DELETE
    // GET /products/list - paginated list with filtering
    // GET /products/id/{id} - get by ID
    // POST /products - create new product
    // PUT /products/set?id={id}&pairs=field:value - update fields
}

```

```
// DELETE /products/id/{id} - delete product  
}
```

1.1.7. Running the Application

Start your application:

```
./mvnw quarkus:dev
```

The application provides: - Swagger UI at <http://localhost:8080/q/swagger-ui/> - Dev UI at <http://localhost:8080/q/dev/>

1.1.8. Testing Your API

Create a product:

```
curl -X POST http://localhost:8080/products \  
-H "Content-Type: application/json" \  
-d '{  
  "sku": "WIDGET-001",  
  "name": "Super Widget",  
  "description": "The best widget ever",  
  "price": 29.99,  
  "active": true  
}'
```

List products:

```
curl "http://localhost:8080/products/list?limit=10&sort=+name"
```

Filter products:

```
curl "http://localhost:8080/products/list?filter=active:true&price:>##20"
```

1.1.9. What You Get Automatically

With this minimal setup, Quantum provides:

Multi-tenancy: Each product is automatically tagged with the creator's DataDomain (tenant, org, owner)

Security: DataDomain filtering ensures users only see their own data by default

Validation: Jakarta Bean Validation runs before persistence

Audit Trail: Automatic createdBy, createdAt, lastUpdatedBy, lastUpdatedDate fields

Consistent APIs: Standard REST patterns across all resources

Query Language: Powerful filtering with the ANTLR-based query syntax

OpenAPI: Automatic API documentation

1.1.10. Next Steps

- Add authentication: [Authentication Guide](#)
- Create sharing rules: [Permissions Guide](#)
- Learn the query language: [Query Language](#)
- See real-world example: [Supply Chain Tutorial](#)

Chapter 2. 2. Platform overview

Problem: Understanding the moving parts and how they fit together.

Why for SaaS: You'll be composing features (tenancy, security, APIs, data) across many modules and services.

How Quantum helps: A cohesive model with clear boundaries and integration points.

Walkthrough: Read the overview and keep it handy as you progress.

2.1. 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.

2.1.1. 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.

Chapter 3. 3. Domain modeling fundamentals

Problem: Modeling your domain effectively for persistence and API exposure.

Why for SaaS: Cross-tenant scale and isolation drive modeling choices (IDs, natural keys, versioning, ownership).

How Quantum helps: Provides base models, annotations, and repository patterns to model tenant-safe entities.

Walkthrough: Build a minimal domain model and persist it.

3.1. 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.

3.1.1. Prefer Annotations for Functional Mapping (recommended)

Starting with this version, you should use annotations to declare a model or resource's Functional Area/Domain and the Action being performed. The legacy `bmFunctionalArea()` and `bmFunctionalDomain()` methods are still supported for backward compatibility in this release, but they will be phased out soon.

- Class-level mapping: use `@FunctionalMapping(area = "<area>", domain = "<domain>")` on your model or resource class.
- Method-level action: use `@FunctionalAction("<ACTION>")` on your JAX-RS resource methods when the action is not implied by the HTTP verb.

Example: model annotated with `FunctionalMapping`

```
import dev.morphia.annotations.Entity;
```

```

import lombok.*;
import lombok.experimental.SuperBuilder;
import com.e2eq.framework.model.persistent.base.BaseModel;
import com.e2eq.framework.annotations.FunctionalMapping;

@Entity
@Data
@NoArgsConstructor
@SuperBuilder
@EqualsAndHashCode(callSuper = true)
@FunctionalMapping(area = "catalog", domain = "product")
public class Product extends BaseModel {
    private String sku;
    private String name;
    // No need to override bmFunctionalArea/bmFunctionalDomain when using
    @FunctionalMapping
}

```

Example: resource method annotated with FunctionalAction

```

import jakarta.ws.rs.*;
import jakarta.ws.rs.core.MediaType;
import com.e2eq.framework.annotations.FunctionalAction;

@Path("/products")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class ProductResource {

    // Action will default from HTTP verb (POST -> CREATE), but you can be explicit:
    @POST
    @FunctionalAction("CREATE")
    public Product create(Product payload) { /* ... */ return payload; }

    // GET will infer VIEW automatically when building the ResourceContext
    @GET
    @Path("/{id}")
    public Product get(@PathParam("id") String id) { /* ... */ return new Product(); }
}

```

How the framework uses these annotations

- **SecurityFilter:** If the matched resource class has `@FunctionalMapping`, it uses area/domain from the annotation. If the method has `@FunctionalAction`, it uses that value; otherwise, it infers the action from the HTTP method (GET=VIEW, POST=CREATE, PUT/PATCH=UPDATE, DELETE=DELETE). If annotations are absent, it falls back to the existing path- and convention-based logic.
- **MorphiaRepo.fillUIActions:** If the model class has `@FunctionalMapping`, its area/domain are

used to resolve allowed UI actions; otherwise, it falls back to the legacy `bmFunctionalArea()/bmFunctionalDomain()` methods.

- `PermissionResource`: When listing functional domains, it prefers `@FunctionalMapping` on entity classes and falls back to `bmFunctionalArea()/bmFunctionalDomain()` when missing.

Migration notes

- Preferred: add `@FunctionalMapping` to each model class (or resource class) and remove the `bmFunctionalArea()/bmFunctionalDomain` overrides.
- Transitional: you can keep the legacy methods; they will be used only if the annotation is not present.
- Future: `bmFunctionalArea` and `bmFunctionalDomain` will be removed in a future release; plan to migrate now.

See also: [Security Annotations: FunctionalMapping and FunctionalAction](#)

3.1.2. 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"; }
}
```

3.1.3. 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
}
```

3.1.4. 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.

3.1.5. 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.

3.1.6. 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.

3.1.7. 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 4. 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 5. 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)
PersistenceAuditEventInterceptor		ReferenceInterceptor	5)

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 6. 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 (path-derived ResourceContext)**: The framework derives area and functionalDomain from REST path segments using the convention: `/{{area}}/{{functionalDomain}}/{{action}}/...`. These are placed on the ResourceContext and consumed by RuleContext. Rule bases typically match on HTTP method and URL patterns; no special headers are required.
- **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) Path-derived rule matching (Permissions)

```
- name: allow-catalog-product-reads
  priority: 300
  match:
    method: [GET]
    url: /Catalog/Product/**
    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

- Path convention: Use leading segments /{area}/{functionalDomain}/{action}/... so the framework can derive ResourceContext reliably. Extra segments after the first three are allowed; only the first three are used to compute area, domain, and action.
- Nonconformant paths: If the path has fewer than three segments, the framework sets an anonymous/default ResourceContext. In practice, rules will typically evaluate to DENY unless there is an explicit allowance for anonymous contexts.
- See also: the Permissions section for rule-base matching and priorities, and the DomainContext/RuleContext section for end-to-end flow.

Chapter 7. 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 8. CompletionTasks and CompletionTaskGroups

CompletionTasks and CompletionTaskGroups provide a simple, persistent way to track a series of work items that need to be completed, either by background processes or external systems. Use them when you need durable progress tracking across restarts and an auditable record of outcomes.

Key models:

- CompletionTask: an individual unit of work with fields like status, timestamps, and optional result/details.
- CompletionTaskGroup: a container that represents a cohort of tasks progressing toward completion.

Model overview:

```
// Individual task
@Entity("completionTask")
public class CompletionTask extends BaseModel {
    public enum Status { PENDING, RUNNING, SUCCESS, FAILED }

    @Reference
    CompletionTaskGroup group; // optional grouping
    String details;           // human-readable context (what/why)
    Status status;            // PENDING -> RUNNING -> (SUCCESS|FAILED)
    Date createdAt;           // when the task was created
    Date completedDate;       // set when terminal (SUCCESS/FAILED)
    String result;            // output, message, or error summary

    @Override public String bmFunctionalArea() { return "TASK"; }
    @Override public String bmFunctionalDomain() { return "COMPLETION_TASK"; }
}

// Group of tasks
@Entity("completionTaskGroup")
public class CompletionTaskGroup extends BaseModel {
    public enum Status { NEW, RUNNING, COMPLETE }

    String description; // e.g., "Onboarding: create resources"
    Status status;       // reflects overall progress of the group
    Date createdAt;      // when the group was created
    Date completedDate;  // when the group finished

    @Override public String bmFunctionalArea() { return "TASK"; }
    @Override public String bmFunctionalDomain() { return "COMPLETION_TASK_GROUP"; }
}
```

Typical lifecycle:

- Create a `CompletionTaskGroup` in `NEW` status.
- Create `N` `CompletionTasks` (status=`PENDING`) referencing the group.
- A worker picks tasks and flips status to `RUNNING`, performs the work, then to `SUCCESS` or `FAILED`, setting `completedDate` and result.
- Periodically update the group:
- If at least one task is `RUNNING` (and none pending), set group status to `RUNNING`.
- When all tasks are terminal (`SUCCESS` or `FAILED`), set group status to `COMPLETE` and `completedDate`.

How to use for tracking a series of things that need to be completed:

- Batch operations: When submitting a batch (e.g., provisioning 100 accounts), create one group and 100 tasks. The UI/API can poll the group to show overall progress and per-item results.
- Multi-step workflows: Represent each step as its own task, or use one task per target resource. Groups help correlate all steps for a single business request.
- Retry/compensation: `FAILED` tasks can be retried by creating new tasks or resetting status to `PENDING` based on your policy. Keep result populated with failure reasons.

Example creation flow:

```
CompletionTaskGroup group = CompletionTaskGroup.builder()
    .description("Catalog import: 250 SKUs")
    .status(CompletionTaskGroup.Status.NEW)
    .createdDate(new Date())
    .build();
completionTaskGroupRepo.save(group);

for (Sku s : skus) {
    CompletionTask t = CompletionTask.builder()
        .group(group)
        .details("Import SKU " + s.code())
        .status(CompletionTask.Status.PENDING)
        .createdDate(new Date())
        .build();
    completionTaskRepo.save(t);
}
```

Example worker progression:

```
// Fetch a PENDING task and execute
CompletionTask t = completionTaskRepo.findOneByStatus(CompletionTask.Status.PENDING);
if (t != null) {
    completionTaskRepo.update(t.getId(), "status", CompletionTask.Status.RUNNING);
    try {
```

```

// ... do work ...
completionTaskRepo.update(t.getId(),
    "status", CompletionTask.Status.SUCCESS,
    "completedDate", new Date(),
    "result", "OK");
} catch (Exception e) {
    completionTaskRepo.update(t.getId(),
        "status", CompletionTask.Status.FAILED,
        "completedDate", new Date(),
        "result", e.getMessage());
}
}

// Periodically recompute group status
List<CompletionTask> tasks = completionTaskRepo.findByGroup(group);
boolean allTerminal = tasks.stream().allMatch(x -> x.getStatus()==SUCCESS || x
.getStatus()==FAILED);
boolean anyRunning = tasks.stream().anyMatch(x -> x.getStatus()==RUNNING);
boolean anyPending = tasks.stream().anyMatch(x -> x.getStatus()==PENDING);

if (allTerminal) {
    completionTaskGroupRepo.update(group.getId(),
        "status", CompletionTaskGroup.Status.COMPLETE,
        "completedDate", new Date());
} else if (anyRunning || (!anyPending && !allTerminal)) {
    completionTaskGroupRepo.update(group.getId(), "status", CompletionTaskGroup.Status
.RUNNING);
}

```

Notes and best practices:

- Keep details short but diagnostic, and store richer context in result.
- Use DataDomain fields for multi-tenant scoping so groups/tasks are isolated per tenant/org as needed.
- Avoid unbounded growth: archive or purge old groups once COMPLETE.
- Consider idempotency keys in details or a custom field to prevent processing the same logical work twice.

Chapter 9. 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 10. 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.

Chapter 11. 4. Multi-tenant model and realm separation

Problem: Isolating data per tenant while enabling selective sharing.

Why for SaaS: Security, compliance, and data residency require strong segregation and auditability.

How Quantum helps: Realm/tenant identifiers, context propagation, transforms, and repository filters.

Walkthrough: Configure realms and mark your models for tenancy.

11.1. Multi-Tenancy Models

Quantum supports multiple multi-tenant models for MongoDB deployments:

11.1.1. 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.

11.1.2. 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.

11.1.3. 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.

Chapter 12. 5. Domain rule context

Problem: Applying business rules based on user, org, account, and tenant context.

Why for SaaS: Entitlements and behavior vary by tenant, role, and plan.

How Quantum helps: Built-in domain rule context to consistently pass identity and policy inputs.

Walkthrough: Inject and use the context in services and repositories.

DomainContext, RuleContext, and DataDomain

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

Chapter 13. 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 14. 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 15. 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 16. 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.

Chapter 17. Resolvers and Variables in Rule Filters

RuleContext can attach FILTERs (not only ALLOW/DENY) to repository queries using rule fields and filter strings. Variables inside those filter strings are populated from:

- PrincipalContext and ResourceContext standard variables: principalId, pAccountId, pTenantId, ownerId, orgRefName, resourceId, action, functionalDomain, area
- AccessListResolver SPI implementations: per-request computed Collections (e.g., customer IDs the caller can access)

Implementation highlights: - AccessListResolver has methods key(), supports(...), resolve(...). Resolvers are injected and invoked for each request; results are published as variables by key. - MorphiaUtils.VariableBundle carries both string variables and object variables (including collections) to the query listener. - The QueryToFilterListener supports IN clauses using a single \${var} inside brackets, expanding Collections/arrays and coercing types (ObjectId, numbers, booleans, dates).

Authoring examples: - Constrain by principal domain:

+

```
orgRefName:${orgRefName} && dataDomain.tenantId:${pTenantId}
```

- Access list resolver for customer visibility:

```
customerId:^([${accessibleCustomerIds}])
```

For the complete query language reference, see [Query Language](#).

DomainContext, RuleContext, and DataDomain

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

Chapter 18. 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 19. 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 20. 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 21. 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.

Chapter 22. Resolvers and Variables in Rule Filters

RuleContext can attach FILTERs (not only ALLOW/DENY) to repository queries using rule fields and filter strings. Variables inside those filter strings are populated from:

- PrincipalContext and ResourceContext standard variables: principalId, pAccountId, pTenantId, ownerId, orgRefName, resourceId, action, functionalDomain, area
- AccessListResolver SPI implementations: per-request computed Collections (e.g., customer IDs the caller can access)

Implementation highlights: - AccessListResolver has methods key(), supports(...), resolve(...). Resolvers are injected and invoked for each request; results are published as variables by key. - MorphiaUtils.VariableBundle carries both string variables and object variables (including collections) to the query listener. - The QueryToFilterListener supports IN clauses using a single \${var} inside brackets, expanding Collections/arrays and coercing types (ObjectId, numbers, booleans, dates).

Authoring examples: - Constrain by principal domain:

+

```
orgRefName:${orgRefName} && dataDomain.tenantId:${pTenantId}
```

- Access list resolver for customer visibility:

```
customerId:^([${accessibleCustomerIds}])
```

For the complete query language reference, see [Query Language](#).

Chapter 23. Concrete example: building and using a resolver

This section shows how to implement a resolver that restricts access to orders by the set of customerIds the current user is allowed to see.

23.1. 1) Implement the SPI

Create a CDI bean that implements `AccessListResolver`. It decides when it applies and returns a Collection of values. The collection can be `ObjectId`, `String`, numbers, etc.

```
import com.e2eq.framework.securityrules.AccessListResolver;
import com.e2eq.framework.model.persistent.base.UnversionedBaseModel;
import com.e2eq.framework.model.securityrules.PrincipalContext;
import com.e2eq.framework.model.securityrules.ResourceContext;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;
import org.bson.types.ObjectId;
import java.util.*;

@ApplicationScoped
public class CustomerAccessResolver implements AccessListResolver {

    @Inject CustomerAccessService service; // your app-specific service

    @Override
    public String key() {
        // This becomes the variable name available to rules: ${accessibleCustomerIds}
        return "accessibleCustomerIds";
    }

    @Override
    public boolean supports(PrincipalContext pctx, ResourceContext rctx,
                           Class<? extends UnversionedBaseModel> modelClass) {
        // Optionally narrow by area/domain/action/model
        return rctx != null && "sales".equalsIgnoreCase(rctx.getArea())
            && "order".equalsIgnoreCase(rctx.getFunctionalDomain());
    }

    @Override
    public Collection<?> resolve(PrincipalContext pctx, ResourceContext rctx,
                                Class<? extends UnversionedBaseModel> modelClass) {
        // Return the set of customer ids for this user; could be ObjectId or String.
        // Example returns strings; the query listener will coerce 24-hex to ObjectId.
        return service.findCustomerIdsForUser(pctx.getUserId());
    }
}
```

Notes: - You can return `List<ObjectId>` directly if you prefer; no coercion needed then. - The resolver runs per request. Cache internally if the computation is expensive.

23.2. 2) How RuleContext uses resolvers

At query time, `RuleContext` discovers all `AccessListResolver` beans and calls `supports(...)`. For those that apply, it invokes `resolve(...)` and publishes the result into the variable bundle under the provided key(). Variables are available to the BI-API query via `${...}`.

Internally this uses `MorphiaUtils.VariableBundle` and `QueryToFilterListener` to carry both strings and typed objects/collections.

23.3. 3) Author a rule that consumes the variable

Given the resolver above, a rule can attach an IN filter to constrain queries:

```
// andFilterString (example)
customerId:^(${accessibleCustomerIds}]
```

When executed: - If `accessibleCustomerIds` is a Collection/array, each element is type-coerced (`ObjectId`, number, date, boolean, or string) and used in `$in`. - If `accessibleCustomerIds` is a comma-separated string, it is split and each token is coerced similarly. - An empty collection results in an empty `$in` (matches none), effectively denying access via filtering, not via ALLOW/DENY.

23.4. 4) End-to-end behavior

- `SecurityFilter` sets `ResourceContext` (area/domain/action) per request.
- `RuleContext` evaluates rules for the principal and resource and gathers resolvers.
- The repository composes filters including the rule-provided IN clause with the access list.
- Only documents whose `customerId` is in the caller's resolved set are returned.

23.5. String literals vs. typed values in resolver variables

When an `AccessListResolver` returns a list of values that will be used in an `IN` clause (for example, `field:^([${var}]]`), the engine attempts to coerce each element to an appropriate type so Mongo/Morphia filters are typed correctly:

- 24-hex string → `ObjectId`
- `true/false` → `Boolean`
- integer → `Long`
- decimal → `Double`
- ISO-8601 datetime → `java.util.Date`

- yyyy-MM-dd → `java.time.LocalDate`
- otherwise → `String`

This works well when your target field is an `ObjectId`, number, or date. However, string fields can contain values that look like other types (for example, a 24-hex string that resembles an `ObjectId`). In those cases you must force "treat as plain string" so no coercion occurs.

To do this, the framework provides a small wrapper type `StringLiteral`. If a resolver returns `StringLiteral` instances, the listener unwraps them to plain `String` values and skips coercion entirely.

23.5.1. Example A: Resolver returns ObjectIds (typed)

```
@ApplicationScoped
public class CustomerAccessResolver implements AccessListResolver {
    public static final ObjectId ID1 = new ObjectId("5f1e1a5e5e5e5e5e5e51");
    public static final ObjectId ID2 = new ObjectId("5f1e1a5e5e5e5e5e5e52");

    @Override public String key() { return "accessibleCustomerIds"; }
    @Override public boolean supports(PrincipalContext p, ResourceContext r, Class<?
extends UnversionedBaseModel> m) {
        return r != null && "sales".equalsIgnoreCase(r.getArea()) && "order"
.equalsIgnoreCase(r.getFunctionalDomain()) && "view".equalsIgnoreCase(r.getAction());
    }
    @Override public Collection<?> resolve(PrincipalContext p, ResourceContext r,
Class<? extends UnversionedBaseModel> m) {
        return java.util.List.of(ID1, ID2); // typed values pass through as-is
    }
}
```

Rule:

```
customerId:^(${accessibleCustomerIds})
```

Result: `$in` with `List<ObjectId>` on `customerId`.

23.5.2. Example B: Resolver returns String literals (force raw strings)

```
@ApplicationScoped
public class CustomerCodeResolver implements AccessListResolver {
    @Override public String key() { return "accessibleCustomerCodes"; }
    @Override public boolean supports(PrincipalContext p, ResourceContext r, Class<?
extends UnversionedBaseModel> m) {
        return r != null && "sales".equalsIgnoreCase(r.getArea()) && "order"
.equalsIgnoreCase(r.getFunctionalDomain()) && "view".equalsIgnoreCase(r.getAction());
    }
    @Override public Collection<?> resolve(PrincipalContext p, ResourceContext r,
```

```

Class<? extends UnversionedBaseModel> m) {
    return java.util.List.of(
        com.e2eq.framework.model.persistent.morphia.StringLiteral.of(
            "5f1e1a5e5e5e5e5e5e5e51"),
        com.e2eq.framework.model.persistent.morphia.StringLiteral.of("CUST-42")
    );
}
}

```

Rule:

```
customerCode:^(${accessibleCustomerCodes}]
```

Result: `$in` with `List<String>` on `customerCode` (even for hex-like strings).

23.5.3. Other types supported

Resolvers can also return numbers, booleans, and dates/datetimes. Already-typed elements (`Number`, `Boolean`, `java.util.Date`, `java.time.LocalDate`, `ObjectId`) are preserved. String elements are heuristically parsed into those types unless wrapped with `StringLiteral`.

Authoring tips:

- Prefer returning already-typed values when you know the target field type.
- Use `StringLiteral` when a value might be misinterpreted (for example, 24-hex or numeric-looking strings).
- For CSV strings published under a variable, the engine splits by comma and applies the same per-element coercion.

Chapter 24. 6. Building RESTful CRUD APIs

Problem: Exposing standardized CRUD endpoints with minimal boilerplate.

Why for SaaS: Consistency across services reduces cognitive load and accelerates delivery.

How Quantum helps: Resource scaffolding, conventions, and helpers for common CRUD.

Walkthrough: Create controllers/resources for your model and wire persistence.

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

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

24.1.1. Base Concepts

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

24.1.2. Example Resource

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

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

24.1.3. Authorization Layers in REST CRUD

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

1) Hard-coded permissions via annotations

- Use standard Jakarta annotations like `@RolesAllowed` (or the framework's `@RoleAllow` if

present) on resource classes or methods to declare role-based checks that must pass before executing an endpoint.

- These checks are fast and decisive. They rely on the caller's roles as established by the current `SecurityIdentity`.

Example:

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

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

2) JWT groups and role mapping

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

3) RuleContext layered authorization (dynamic policies)

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

4) Quarkus `SecurityIdentity` and `SecurityFilter`

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

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

24.1.4. Querying

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

24.1.5. Responses and Schemas

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

24.1.6. Error Handling

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

24.1.7. 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}`

24.2. 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}
```

24.3. 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.

24.4. IN lists

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

24.5. 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
```

24.6. 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
```

24.7. 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 25. 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.

25.1. 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 \leq `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"
```

25.2. 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,principalId,description&skipHeaderRow=true&fieldSeparator=,&quoteChar=\"&quotingStrategy=QUOTE_WHERE_ESSENTIAL&charsetEncoding=UTF-8-without-BOM"
```

25.3. Import with preview sessions

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

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

Notes and constraints:

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

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

Authentication and Authorization

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

Chapter 26. 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 27. 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 28. 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 29. 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 30. 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 31. 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 32. 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 33. CredentialUserIdPassword model and DomainContext

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

What the credential model represents

userId

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

subject

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

description, emailOfResponsibleParty

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

domainContext

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

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

roles

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

issuer

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

passwordHash, hashingAlgorithm

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

forceChangePassword

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

lastUpdate

Timestamp for auditing and token invalidation strategies.

area2RealmOverrides

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

realmRegEx

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

impersonateFilterScript

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

authProviderName

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

How DomainContext selects the realm for REST calls

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

Chapter 34. Quarkus OIDC out-of-the-box and integrating with common IdPs

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

What the Quarkus OIDC extension provides

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

Integrating with common providers

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

OIDC vs OAuth vs OpenID (terminology and evolution)

OAuth 2.0

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

OpenID (OpenID 1.x/2.0)

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

OpenID Connect (OIDC)

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

underneath.

Summary

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

OIDC and SAML in relation to SSO

SAML (Security Assertion Markup Language)

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

OIDC

JSON/REST-oriented, uses JWTs, and is well-suited for modern SPAs and APIs.

Relationship: * Both enable SSO and federation across identity providers and service providers. * Many enterprise IdPs support both; OIDC is generally simpler for APIs and modern web stacks, while SAML is entrenched in legacy/enterprise SSO.

Bridging: * Gateways or identity brokers can translate SAML assertions to OIDC tokens and vice versa, allowing gradual migration.

Common customer IdP models and OIDC integration patterns

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

Chapter 35. Authorization via RuleContext

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

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

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

Chapter 36. 7. Query language and filtering

Problem: Powerful, safe, and consistent querying across collections.

Why for SaaS: Tenants need flexible reporting and filtration without bespoke endpoints.

How Quantum helps: A uniform query language layer with server-side enforcement.

Walkthrough: Add query endpoints and test filters/security.

36.1. Query Language Reference

Quantum uses an ANTLR-based query language (BIAPIQuery.g4) for filtering, searching, and constraining data across all REST endpoints. This single, consistent syntax works everywhere: list APIs, permission rules, and access resolvers.

36.1.1. Basic Syntax

Operators

Operator	Symbol	Example
Equals	:	name:"Widget"
Not equals	:!	status:! "DELETED"
Less than	:<	price:<##100
Greater than	:>	quantity:>#50
Less than or equal	:≤	createdAt:≤2024-12-31
Greater than or equal	:>=	updatedAt:>=2024-01-01T00:00:00Z
Field exists	:~	description:~
In list	:^	status:^["ACTIVE","PENDING"]
Not in list	:!^	status:!^["DELETED","ARCHIVED"]

Value Types

Type	Prefix	Example
String	none or "..."	name:widget or name:"Super Widget"
Number (integer)	#	quantity:#42
Number (decimal)	##	price:##19.99
Date	none	shipDate:2024-12-25

Type	Prefix	Example
DateTime	none	createdAt:2024-12-25T10:30:00Z
Boolean	none	active:true
Null	none	description:null
ObjectId	none	id:507f1f77bcf86cd799439011
Reference	@@	parentId:@@507f1f77bcf86cd799439011
Variable	\${...}	ownerId:\${principalId}

Logical Operators

Operator	Symbol	Example
AND	&&	active:true && price:>##10
OR		status:"ACTIVE" status:"PENDING"
NOT	!!	!(price:<##5)
Grouping	()	(active:true featured:true) && price:>##0

36.1.2. Common Patterns

String Matching

```
# Exact match
name:"Super Widget"

# Wildcard matching
name:*widget*      # contains "widget"
name:widget*       # starts with "widget"
name:*widget       # ends with "widget"
name:w?dget        # single character wildcard

# Case sensitivity (depends on database collation)
name:"WIDGET"       # may or may not match "widget"
```

Numeric Ranges

```
# Price between 10 and 100
price:>=##10 && price:<=##100

# Quantity greater than 0
quantity:>#0

# Exact count
itemCount:#5
```

Date and Time Queries

```
# Orders from today
createdAt:>=2024-12-25

# Orders from last week
createdAt:>=2024-12-18 && createdAt:<2024-12-25

# Specific timestamp
updatedAt:2024-12-25T14:30:00Z

# Orders modified this year
updatedAt:>=2024-01-01T00:00:00Z
```

List Membership

```
# Status in specific values (IN)
status:^[ "ACTIVE", "PENDING", "PROCESSING" ]

# Exclude statuses (NOT IN)
status:!^[ "DELETED", "ARCHIVED" ]

# User IDs from a list (IN)
ownerId:^[ "user1", "user2", "user3" ]

# Exclude specific users (NOT IN)
ownerId:!^[ "user1", "user2" ]

# ObjectId list (IN)
categoryId:^[ @507f1f77bcf86cd799439011, @507f1f77bcf86cd799439012 ]

# ObjectId list (NOT IN)
categoryId:!^[ @507f1f77bcf86cd799439011, @507f1f77bcf86cd799439012 ]

# Mixed types (coerced automatically)
priority:^[ #1, #2, #3 ]

# Using variables (CSV expansion supported by access resolvers)
customerId:!^[ ${accessibleCustomerIds} ]
```

Null and Existence Checks

```
# Field has any value (not null)
description:~

# Field is null
description:null
```

```
# Field is not null
description:!null
```

```
# Field exists and is not empty string
description:~ && description:!""
```

36.1.3. Advanced Examples

Complex Business Logic

```
# Active products under $50 OR featured products at any price
(active:true && price:<##50) || featured:true
```

```
# Orders needing attention: overdue OR high-value pending
(dueDate:<2024-12-25 && status!="COMPLETED") ||
(status:"PENDING" && totalAmount:>##1000)
```

```
# Products with inventory issues
(quantity:<=#5 && reorderPoint:>#5) || stockStatus:"OUT_OF_STOCK"
```

Multi-tenant Filtering

```
# User's own records
dataDomain.ownerId:${principalId}
```

```
# Organization-wide access
dataDomain.orgRefName:${orgRefName}
```

```
# Tenant-scoped with public sharing
dataDomain.tenantId:${pTenantId} || dataDomain.orgRefName:"PUBLIC"
```

Audit and Compliance

```
# Records modified by specific user
auditInfo.lastUpdatedBy:"john.doe"
```

```
# Changes in date range
auditInfo.lastUpdatedDate:>=2024-12-01 &&
auditInfo.lastUpdatedDate:<2024-12-31
```

```
# Created vs modified
auditInfo.createdDate:auditInfo.lastUpdatedDate # never modified
auditInfo.createdDate:!auditInfo.lastUpdatedDate # has been modified
```

36.1.4. Variables in Filters

Variables are resolved from the current security context and can be used in permission rules and access resolvers.

Standard Variables

Variable	Description
<code>\${principalId}</code>	Current user's ID
<code>\${pTenantId}</code>	Principal's tenant ID
<code>\${pAccountId}</code>	Principal's account ID
<code>\${pOrgRefName}</code>	Principal's organization
<code>\${realm}</code>	Current realm/database
<code>\${area}</code>	Current functional area
<code>\${functionalDomain}</code>	Current functional domain
<code>\${action}</code>	Current action (CREATE, UPDATE, etc.)

Custom Variables from Access Resolvers

```
// In your AccessListResolver
@Override
public String key() {
    return "accessibleCustomerIds"; // becomes ${accessibleCustomerIds}
}

@Override
public Collection<?> resolve(...) {
    return Arrays.asList("CUST001", "CUST002", "CUST003");
}
```

```
# Use in filter
customerId:^(${accessibleCustomerIds}]
```

36.1.5. Performance Tips

Efficient Queries

```
# Good: Use indexed fields first
status:"ACTIVE" && createdAt:>=2024-01-01

# Better: Combine with specific values
status:"ACTIVE" && ownerId:${principalId} && createdAt:>=2024-01-01
```



```
# Avoid: Leading wildcards on large collections
name:*widget # can be slow on millions of records
```

Projection for Large Objects

```
# In REST calls, limit returned fields
GET /products/list?filter=active:true&projection=+id,+name,+price,-description
```

36.1.6. Integration with REST APIs

List Endpoints

```
# Basic filtering
GET /products/list?filter=active:true

# With sorting and pagination
GET /products/list?filter=price:>##10&sort=-createdDate&skip=20&limit=10

# Complex filter with projection
GET
/orders/list?filter=(status:"PENDING"||status:"PROCESSING")&&totalAmount:>##100&projection=+id,+status,+totalAmount,+customerName
```

Permission Rules

```
- name: user-own-records
  priority: 300
  match:
    method: [GET]
    url: /api/**
  effect: ALLOW
  andFilterString: "dataDomain.ownerId:${principalId}"
```

Access Resolvers

```
// Resolver returns customer IDs user can access
public Collection<?> resolve(...) {
    return customerService.getAccessibleIds(principalId);
}

// Used in permission rule
andFilterString: "customerId:^(${accessibleCustomerIds})"
```

36.1.7. Error Handling

Common syntax errors and solutions:

```
# Wrong: Missing quotes for multi-word strings
name:Super Widget
# Right:
name:"Super Widget"

# Wrong: Incorrect number prefix
price:19.99
# Right:
price:##19.99

# Wrong: Invalid date format
createdDate:12/25/2024
# Right:
createdDate:2024-12-25

# Wrong: Unbalanced parentheses
(active:true && price:>##10
# Right:
(active:true && price:>##10)
```

36.1.8. See Also

- [REST CRUD Querying](#)
- [Permission Rules](#)
- [Access Resolvers](#)

Chapter 37. 8. Authentication, permissions, and annotations

Problem: Securely authenticating users and enforcing fine-grained authorization.

Why for SaaS: Multi-tenant apps must ensure tenant isolation and principle-of-least-privilege.

How Quantum helps: JWT integration, permissions model, and security annotations.

Walkthrough: Protect your APIs and add role/permission checks.

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 38. 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 39. 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 40. 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 41. 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 42. 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 43. 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 44. 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 45. 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 46. 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 47. 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 48. Introduction: Layered Enforcement Overview

Quantum evaluates "can this identity do X?" through three complementary layers. Understanding them in order helps you pick the right tool for the job and combine them safely:

1. REST API annotations (top layer, code-level)

- What: JAX-RS/Jakarta Security annotations on resource methods, for example, `@RolesAllowed("ADMIN")`, `@PermitAll`, `@DenyAll`, `@Authenticated`.
- Purpose: Coarse-grained, immediate gates right at the endpoint. Ideal for baseline protections (for example, only ADMIN may call `/admin/**`) and for non-dynamic constraints that rarely change.
- Pros: Simple, fast, visible in code reviews.
- Cons: Hard-coded; changing access requires a code change, build, and deploy. No data-aware scoping (for example, cannot express tenant/domain filters).

2. Feature flags (exposure and variants)

- What: Turn capabilities on/off per environment or cohort and select variants (A/B, multivariate). See "Feature Flags, Variants, and Target Rules" below.
- Purpose: Control who even sees or can reach a capability during rollout (by tenant, role, geography, plan), independently of authorization. Flags answer "is the feature ON and which variant?"; they do not by themselves prove the caller is authorized.
- Pros: Reversible, environment-aware, safe rollout and experimentation.
- Cons: Not a substitute for authorization; must be paired with roles/permission rules for enforcement.

3. Permission Rules with DataDomain filtering (fine-grained, dynamic)

- What: Declarative rule bases that match URL, method, SecurityURLHeaders, and SecurityURLBody, and decide ALLOW/DENY. ALLOWs can contribute DataDomain constraints applied by repositories. See sections "Key Concepts", "Matching Algorithm", and examples below.
- Purpose: Express least-privilege, data-aware policies that evolve without code changes (data-driven authoring).
- Pros: Dynamic, auditable, supports role checks plus attribute predicates and domain scoping.
- Cons: Requires governance of rulebases and careful priority management.

How roles, Functional Areas, and Functional Domains fit in

- Roles: Used by both layers (1) and (3). Annotations directly reference roles; rules can require `rolesAny`/`rolesAll`. Effective roles are resolved by merging IdP roles with roles on the user record; see "How roles are defined for an identity" below.
- Functional Area/Domain: Derived from the URL convention

`/{{area}}/{{functionalDomain}}/{{action}}` as parsed by `SecurityFilter.determineResourceContext`. Author policies using these fields to target business capabilities rather than raw URLs or ad-hoc headers.

- **DataDomain:** When rules ALLOW an action, they can attach data-scope filters (tenant/org/owner) so downstream reads/writes are constrained to the caller's domain.

Choosing the right approach

- Use annotations for stable, coarse gates you want visible in code (for example, admin-only endpoints, health endpoints with `@PermitAll`).
- Use feature flags to manage rollout/exposure and variants across environments and cohorts.
- Use permission rules to encode fine-grained, data-aware authorization and to evolve policy without redeploying.

Compare and contrast

- Annotation-based controls are compile-time and hard-code policy into the service; changing them requires code changes.
- Permission Rules and the Rule Language are data-driven and user-changeable (with proper governance), enabling rapid, auditable policy changes and DataDomain scoping.
- In practice: apply annotations as the first gate, evaluate feature flags to determine exposure/variant, then evaluate permission rules to decide ALLOW/DENY and attach scopes. This layered approach yields both safety and agility.

Chapter 49. 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 adds human information such as first name, last name, email address, phone number and provides a linkage back to 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 (for example, `X-Tenant-Id`). Note: functional area/domain/action are derived from the URL, not headers.
 - `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 50. Rule Structure (Illustrative)

```
- name: allow-public-reads
  priority: 100
  match:
    method: [GET]
    url: /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. Functional area/domain/action are parsed from the URL convention `/area/functionalDomain/action` (see `SecurityFilter.determineResourceContext`).
- Body predicates (SecurityURLBody) can be expressed similarly as JSONPath-like constraints:

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

Chapter 51. 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 52. 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 53. Grant-based vs Deny-based Rule Sets

Grant-based rule sets start with a default decision of DENY and then incrementally add ALLOW scenarios through explicit rules. This model is fail-safe by default: any URL, action, or functional area that does not have a matching ALLOW rule remains inaccessible. As new endpoints or capabilities are added to the system, users will not gain access until an explicit ALLOW is authored. This is the recommended posture for security-sensitive systems and multi-tenant platforms.

Deny-based rule sets start with a default decision of ALLOW and then add DENY scenarios to carve away disallowed cases. In this model, new functionality is exposed by default unless a DENY is added. While convenient during rapid prototyping, this posture risks accidental exposure as the surface area grows.

Practical implications:

- Change management: Grant-based requires adding ALLOWs when shipping new features; Deny-based requires remembering to add new DENYs.
- Auditability: Grant-based policies make it easy to enumerate what is permitted; Deny-based requires proving the absence of permissive gaps.
- Safety: In merge conflicts or partial deployments, Grant-based tends to fail closed (DENY), which is usually safer.

Example defaults:

- Grant-based (recommended):

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

- Deny-based (use with caution):

```
- name: default-allow
  priority: 10000
  match: {}
  effect: ALLOW
```

Tip: Even in a deny-based set, author low-number DENY rules for critical protections. In most production systems, prefer the grant-based model and layer specific ALLOWs for each capability.

Chapter 54. Feature Flags, Variants, and Target Rules

Feature flags complement permission rules by controlling whether a capability is active for a given principal, cohort, or environment. Permissions answer “may this identity perform this action?”; feature flags answer “is this capability turned on, and which variant applies?” Use them together to achieve safe rollouts and fine-grained authorization.

Model reference: `com.e2eq.framework.model.general.FeatureFlag` with key fields:

- `enabled`: master on/off
- `type`: `BOOLEAN` or `MULTIVARIATE`
- `variants`: list of variant keys for multivariate experiments
- `targetRules`: cohort targeting rules
- `environment`: e.g., `dev`, `staging`, `prod`
- `jsonConfiguration`: arbitrary configuration for the feature (e.g., rollout %, UI copy, limits)

Example: Boolean flag for a new export API with environment-specific targeting

```
{
  "refName": "EXPORT_API",
  "description": "Enable CSV export endpoint",
  "enabled": true,
  "type": "BOOLEAN",
  "environment": "prod",
  "targetRules": [
    { "attribute": "role", "operator": "equals", "values": ["BETA"] },
    { "attribute": "tenantId", "operator": "in", "values": ["T100", "T200"] }
  ],
  "jsonConfiguration": { "rateLimitPerMin": 60 }
}
```

Example: Multivariate flag to roll out Search v2 to 10% of users and all members of a beta role

```
{
  "refName": "SEARCH_V2",
  "description": "New search implementation",
  "enabled": true,
  "type": "MULTIVARIATE",
  "variants": ["control", "v2"],
  "environment": "prod",
  "targetRules": [
    { "attribute": "role", "operator": "equals", "values": ["BETA"], "variant": "v2" },
    { "attribute": "userId", "operator": "hashMod", "values": ["10"], "variant": "v2" }
  ],
}
```

```

}
],
"jsonConfiguration": { "defaultVariant": "control" }
}

```

Notes on TargetRules:

- attribute: a property from identity/userProfile (e.g., userId, role, tenantId, location, plan).
- operator: equals, in, contains, startsWith, regex, or domain-specific operators like hashMod for percentage rollouts.
- values: comparison values; semantics depend on operator.
- variant: when type is MULTIVARIATE, selects which variant applies when the rule matches.

How feature flags complement Permission Rule Context:

- The evaluation of a request can enrich the Rule Context (SecurityURLHeaders/Body or userProfile) with resolved feature flags and variants (e.g., userProfile.features["SEARCH_V2"] = "v2").
- Permission rules can then require a feature to be present before ALLOWing an action:

```

- name: allow-export-when-flag-on
  priority: 300
  match:
    method: [GET]
    url: /Reports/Export/**
    # Example predicate that assumes features are surfaced in userProfile
    userProfile.features.EXPORT_API: [true]
  rolesAny: [ADMIN, REPORTER]
  effect: ALLOW

```

Alternatively, systems may surface feature decisions via headers (e.g., X-Feature-SEARCH_V2: v2) so that SecurityURLHeaders can match directly.

Business usage examples for TargetRules and their correlation to Permission Rules:

- Progressive rollout by tenant TargetRule tenantId in [T100, T200] → Permission adds ALLOW for endpoints guarded by that flag so only those tenants can call them during rollout.
 - Role-based beta access: TargetRule role equals BETA → Permission requires both the BETA feature flag and standard role checks (e.g., USER/ADMIN) to ALLOW sensitive actions.
 - Plan/entitlement tiers: TargetRule plan in [Pro, Enterprise] → Permission rules enforce additional data-domain constraints (e.g., export size limits) while the flag simply turns the feature on for eligible plans.

Guidance: Feature Flags vs Permission Rules

- Put into Feature Flags:

- Gradual, reversible rollouts; A/B or multivariate experiments; UI/behavior switches.
- Environment gates (dev/staging/prod) and cohort targeting (tenants, beta users, geography).
- Non-security configuration values in jsonConfiguration (limits, thresholds, copy) that do not change who is authorized.
- Put into Permission Rules:
 - Durable authorization logic: roles, identities, functional area/domain/action, and DataDomain constraints.
 - Compliance and least-privilege decisions where fail-closed behavior is required.
 - Enforcement that remains valid after a feature is fully launched (even when the flag is removed).

Recommendation

Use a grant-based permission posture (default DENY) and let feature flags decide which cohorts even see or can reach new capabilities. Then author explicit ALLOW rules for those capabilities, conditioned on both role and feature presence.

Chapter 55. 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 56. 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.

56.1. How roles are defined for an identity (role sources and resolution)

Quantum composes the effective roles for a request by merging:

- Roles from the identity provider (JWT/`SecurityIdentity`)
- Roles configured on the user record (`CredentialUserIdPassword.roles`)

Source details:

- Identity Provider (JWT): roles commonly arrive via standard claims (for example, `groups`, `roles`, or provider-specific fields). Quarkus maps these into `SecurityIdentity.getRoles()`. In multi-realm setups, the realm in `X-Realm` can scope lookups but does not alter what the JWT asserts.
- Quantum user record: `com.e2eq.framework.model.security.CredentialUserIdPassword` has a `String[] roles` field stored per realm. This can be administered by Quantum to grant platform- or tenant-level roles.

Merge semantics (current implementation):

- Union: the effective role set is the union of JWT roles and `CredentialUserIdPassword.roles`. If either source is empty, the other source defines the set.
- Fallback: when neither source yields roles, the framework defaults to `ANONYMOUS`.
- Where implemented: `SecurityFilter.determinePrincipalContext` builds `PrincipalContext` with the merged roles.

Realm considerations:

- The user record is looked up by subject or userId in the active realm (default or `X-Realm`). If a realm override is provided, it is validated with `CredentialUserIdPassword.realmRegex`.
- Roles stored in a user record are realm-specific; JWT roles are whatever the IdP asserts for the token.

Operating models:

- Quantum-managed roles:
 - IdP authenticates the user (subject, username). Authorization is primarily driven by roles

stored in `CredentialUserIdPassword.roles`.

- Use when you want central, auditable role assignment within Quantum, independent of IdP groups.
- IdP-managed roles:
 - IdP carries authoritative roles/groups in the JWT. Keep `CredentialUserIdPassword.roles` minimal or empty.
 - Use when enterprises require IdP as the source of truth for access groups.
- Hybrid (recommended in many deployments):
 - Effective roles = JWT roles \cup `CredentialUserIdPassword.roles`.
 - Use JWT for enterprise groups (for example, `DEPT_SALES`, `ORG_ADMIN`) and Quantum roles for app-specific grants (for example, `REPORT_EXPORTER`, `BETA`).
 - This avoids IdP churn for application-local concerns while respecting org policies.

Examples:

- JWT-only:
 - `JWT.groups = [USER, REPORTER]`; user record roles = []
 - Effective roles = [USER, REPORTER]
- Quantum-only:
 - `JWT.groups = []`; user record roles = [USER, ADMIN]
 - Effective roles = [USER, ADMIN]
- Hybrid union:
 - `JWT.groups = [USER]`; user record roles = [BETA, REPORT_EXPORTER]
 - Effective roles = [USER, BETA, REPORT_EXPORTER]

Guidance and best practices:

- Keep role names stable and environment-agnostic; use realms/permissions to scope where needed.
- Avoid overloading roles for feature rollout; use Feature Flags for rollout and variants, and roles for durable authorization.
- When IdP is authoritative, ensure consistent claim mapping so `SecurityIdentity.getRoles()` contains the expected values; commonly via `groups` claim in JWT.
- Use grant-based permission rules and require the minimal set of roles (`rolesAny/rolesAll`) needed for each capability.

Cross-references:

- User model: `com.e2eq.framework.model.security.CredentialUserIdPassword.roles`
- Context: `com.e2eq.framework.model.securityrules.PrincipalContext.getRoles()`
- Filter logic: `com.e2eq.framework.rest.filters.SecurityFilter.determinePrincipalContext`

Chapter 57. Example Scenarios

1. Public catalog browsing

- Request: GET /Catalog/Products/VIEW?search=widgets
- 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 /Collaboration/Shipments/UPDATE
- Headers: 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 58. 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 59. 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 (for example, 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 60. 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 61. 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.

61.1. 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
  }
]
}

```

61.2. 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

61.3. 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=+refName&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"
```

61.4. 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 (for example, 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 62. Realm override (X-Realm) and Impersonation (X-Impersonate)

This section explains how to use the request headers X-Realm and X-Impersonate-* alongside permission rule bases. These headers influence which realm (database) a request operates against and, in the case of impersonation, which identity's roles are evaluated by the rule engine.

62.1. What they do (at a glance)

- X-Realm: Overrides the target realm (MongoDB database) used by repositories for this request. Your own identity and roles remain the same; only the data context (tenant/realm) changes for this call. This lets you “switch tenants” at the database level in deployments that use the one-tenant-per-database model.
- X-Impersonate-Subject or X-Impersonate-UserId: Causes the request to run as another identity. The effective permissions become those of the impersonated identity (potentially more or less than your own). This is analogous to `sudo` on Unix or to “simulate a user/role” for troubleshooting.

Only one of X-Impersonate-Subject or X-Impersonate-UserId may be supplied per request. Supplying both results in a 400/IllegalArgumentException.

62.2. How the headers integrate with permission evaluation

- Rule matching and effects (ALLOW/DENY) still follow the standard algorithm described earlier.
- With X-Realm (no impersonation):
 - The `PrincipalContext.defaultRealm` is set to the header value (after validation), and repositories operate in that realm.
 - Your own roles and identity remain intact; the rule base is evaluated for your identity and roles but in the specified realm's data context.
- With impersonation:
 - The `PrincipalContext` is rebuilt from the impersonated user's credential. The effective roles used by the rule engine include the impersonated user's roles; the platform also merges in the caller's security roles from `Quarkus SecurityIdentity`. This means permissions can be a superset; design policy rules accordingly.
 - The effective realm for the request is set to the impersonated user's default realm (not the X-Realm header). If you passed X-Realm, it is still validated (see below) but not used to override the impersonated default realm in the current implementation.

62.3. Required credential configuration (CredentialUserIdPassword)

Two fields on `CredentialUserIdPassword` govern whether a user may use these headers:

- `realmRegEx` (for `X-Realm`):
 - A wildcard pattern ("*" matches any sequence; case-insensitive) listing the realms a user is allowed to target with `X-Realm`.
 - If `X-Realm` is present but `realmRegEx` is null/blank or does not match the requested realm, the server returns 403 Forbidden.
- Examples:
 - "*" → allow any realm
 - "acme-*" → allow realms that start with acme-
 - "dev|stage|prod" is not supported as-is; use wildcards like "dev*" and "stage*" or a combined pattern like "(dev|stage|prod)" only if you store a true regex. The current validator replaces "with "." and matches case-insensitively.
- `impersonateFilterScript` (for `X-Impersonate-*`):
 - A JavaScript snippet executed by the server (GraalVM) that must return a boolean. It receives three variables: `username` (the caller's subject), `userId` (caller's `userId`), and `realm` (the requested realm or current DB name).
 - If the script evaluates to false, the server returns 403 Forbidden for impersonation.
 - If the script is missing (null) and you attempt impersonation, the server rejects the request with `400/IllegalArgumentException`.

Example impersonation script (allow only company admins to impersonate in dev realms):

```
// username = caller's subject, userId = caller's userId, realm = requested realm (or current)
(username.endsWith('@acme.com') && realm.startsWith('dev-'))
```

Tip: Manage these two fields via your auth provider's admin APIs or directly through `CredentialRepo` in controlled environments.

62.4. End-to-end behavior from SecurityFilter (reference)

The `SecurityFilter` constructs the `PrincipalContext/ResourceContext` before rule evaluation:

- `X-Realm` is read and, if present, validated against the caller's `credential.realmRegEx`.
- If impersonation headers are present:
 - The caller's `credential.impersonateFilterScript` is executed. If it returns true, the impersonated user's credential is loaded and used to build the `PrincipalContext`.
 - The final `PrincipalContext` carries the impersonated user's `defaultRealm` and roles (merged with the caller's `SecurityIdentity` roles), and may copy `area2RealmOverrides` from the impersonated

credential. - Without impersonation, the PrincipalContext is built from the caller's credential; X-Realm, when valid, sets the defaultRealm for this request.

62.5. Practical differences and use cases

- Realm override (X-Realm):
- Who you are does not change; only where you act changes. Your permissions (as determined by policies attached to your identity/roles) are applied against data in the specified realm.
- Use cases:
- Multi-tenant admin tooling that needs to inspect or repair data in customer realms.
- Reporting or backfills where the same service is pointed at different tenant databases per request.
- Impersonation (X-Impersonate-*):
- Who you are (for authorization purposes) changes. You act with the impersonated identity's permissions; depending on your configuration, additional caller roles may be merged.
- Use cases:
- Temporary elevation to an admin identity (sudo-like) for break-glass operations.
- Simulate what a given role/identity can see/do for troubleshooting or customer support.

Caveats: - Never set a permissive impersonateFilterScript in production. Keep it restrictive and auditable. - When using both X-Realm and impersonation in one call, be aware that the effective realm will be the impersonated user's default realm; X-Realm is not applied in the impersonation branch in the current implementation. - realmRegex must be populated for any user who needs realm override; leaving it blank effectively disables X-Realm for that user.

62.6. Examples

- List policies in a different realm using your own identity

```
curl -H "Authorization: Bearer $JWT" \  
  -H "X-Realm: acme-prod" \  
  "https://host/api/security/permission/policies/list?limit=20&sort=+refName"
```

- Simulate another user by subject while staying in their default realm

```
curl -H "Authorization: Bearer $JWT" \  
  -H "X-Impersonate-Subject: 3d8f4e7b-...-idp-subject" \  
  "https://host/api/security/permission/policies/list?limit=20"
```

- Attempt impersonation with a realm hint (validated by script; effective realm = impersonated default)

```
curl -H "Authorization: Bearer $JWT" \  
-H "X-Realm: dev-acme" \  
-H "X-Impersonate-UserId: tenant-admin" \  
"https://host/api/security/permission/policies/list?limit=20"
```

Security outcomes in all cases continue to be driven by your rule bases (Policy rules) matched against the effective `PrincipalContext` and `ResourceContext`.

Chapter 63. Data domain assignment on create: DomainContext and DataDomainPolicy

This section explains how Quantum decides which dataDomain is stamped on newly created records, why this decision is necessary in a multi-tenant system, what the default behavior is, and how you can override it globally or per Functional Area / Functional Domain. It also describes the DataDomainResolver interface and the default implementation provided by the framework.

63.1. The problem this solves (and why it matters)

In a multi-tenant platform you must ensure each new record is written to the correct data partition so later reads/updates can be scoped safely. If the dataDomain is wrong or missing, you risk leaking data across tenants or making your own data inaccessible due to mis-scoping.

Historically, Quantum set the dataDomain of new entities to match the creator's credential (i.e., the principal's DomainContext → DataDomain). That default is sensible in many cases, but real systems often need more specific behavior per business area or type. For example: - You may centralize HR records in a single org-level domain regardless of who created them. - Sales invoices for EU customers must live under an EU data segment. - A specific product area might always write into a shared catalog domain separate from the author's tenant.

These needs require a simple, deterministic way to override the default per Functional Area and/or Functional Domain.

63.2. Key concepts recap: DomainContext and DataDomain

DomainContext (on credentials/realms)

captures the principal's scoping defaults (realm, org/account/tenant identifiers, data segment). At request time this is materialized into a DataDomain.

DataDomain

is what gets stamped onto persisted entities and later used by repositories to constrain queries and updates.

If you do nothing, new records inherit the principal's DataDomain.

63.3. The default policy (do nothing and it works)

Out of the box, Quantum preserves the existing behavior: if no policy is configured, the resolver falls back to the authenticated principal's DataDomain. This guarantees compatibility with existing applications.

Concretely: - ValidationInterceptor checks if an entity being persisted lacks a dataDomain. - If

missing, it calls `DataDomainResolver.resolveForCreate(area, domain)`. - The `DefaultDataDomainResolver` first looks for overrides (credential-attached or global); if none match, it returns the principal's `DataDomain` from the current `SecurityContext`.

63.4. Policy scopes: principal-attached vs. global

You can define overrides at two levels: - Principal-attached (per credential): attach a `DataDomainPolicy` to a `CredentialUserIdPassword`. The `SecurityFilter` places this policy into the `PrincipalContext`, so it applies only to records created by that principal. This is useful for VIP service accounts or specific partners. - Global policy: an application-wide `DataDomainPolicy` provided by `GlobalDataDomainPolicyProvider`. If present, this applies when the principal has no specific override for the matching area/domain.

Precedence: principal-attached policy wins over global policy; if neither applies, fall back to the principal's credential domain.

63.5. The policy map and matching

A `DataDomainPolicy` is a small map of rules: `Map<String, DataDomainPolicyEntry> policyEntries`, keyed by "<FunctionalArea>:<FunctionalDomain>" with support for "*" wildcards. The resolver evaluates keys in this order:

1. area:domain (most specific)
2. area:*
3. *:domain
4. : (global catch-all)
5. Fallback to principal's domain if no entry yields a value

Each `DataDomainPolicyEntry` has a `resolutionMode`: - `FROM_CREDENTIAL` (default): use the principal's credential domain (i.e., the historical behavior). - `FIXED`: use the first `DataDomain` listed in `dataDomains` on the entry.

Example policy definitions (illustrative JSON):

```
{
  "policyEntries": {
    "Sales:Invoice": { "resolutionMode": "FIXED", "dataDomains": [ {"orgRefName":
"ACME", "tenantId": "eu-1", "dataSegment": "INVOICE"} ] },
    "Sales:*": { "resolutionMode": "FROM_CREDENTIAL" },
    "*:HR": { "resolutionMode": "FIXED", "dataDomains": [ {"orgRefName":
"GLOBAL", "tenantId": "hr", "dataSegment": "HR"} ] },
    "*:*": { "resolutionMode": "FROM_CREDENTIAL" }
  }
}
```

Behavior of the above: - Sales:Invoice records always go to the fixed EU invoices domain. - Any

other Sales:* creation uses the creator's credential domain. - All HR records go to a central HR domain. - Otherwise, default to the creator's domain.

63.6. How the resolver works

Interfaces and default implementation:

```
public interface DataDomainResolver {
    DataDomain resolveForCreate(String functionalArea, String functionalDomain);
}

@ApplicationScoped
public class DefaultDataDomainResolver implements DataDomainResolver {
    @Inject GlobalDataDomainPolicyProvider globalPolicyProvider;
    public DataDomain resolveForCreate(String area, String domain) {
        DataDomain principalDD = SecurityContext.getPrincipalDataDomain()
            .orElseThrow(() -> new IllegalStateException("Principal context not providing a data domain"));
        List<String> keys = List.of(areaOrStar(area)+":"+areaOrStar(domain), areaOrStar(area)+":*", ".*:"+areaOrStar(domain), ".*:*");
        // 1) principal attached policy from PrincipalContext
        DataDomain fromPrincipal = resolveFrom(policyFromPrincipal(), keys, principalDD);
        if (fromPrincipal != null) return fromPrincipal;
        // 2) global policy
        DataDomain fromGlobal = resolveFrom(globalPolicyProvider.getPolicy().orElse(null), keys, principalDD);
        if (fromGlobal != null) return fromGlobal;
        // 3) default fallback
        return principalDD;
    }
}
```

Integration point: - ValidationInterceptor injects DataDomainResolver and calls it in prePersist when an entity's dataDomain is null. - SecurityFilter propagates a principal's attached DataDomainPolicy (if any) into the PrincipalContext so the resolver can see it.

63.7. When would you want a non-global policy?

Here are a few concrete scenarios: - Centralized HR: All HR Employee records are written to a shared HR domain regardless of the team creating them. This supports a shared-service HR model without duplicating HR data per tenant. - Regulated invoices: In the Sales:Invoice domain for EU, you must write under a specific EU tenantId/dataSegment to satisfy data residency. Other Sales domains can keep default behavior. - Shared catalog: The Catalog:Item domain is a cross-tenant shared catalog maintained by a core team. Writes should go to a canonical catalog domain even when initiated by tenant-specific users. - VIP account override: A particular integration user should always write to a staging domain for testing purposes, while all others use defaults. Attach a small policy to just that credential.

63.8. Relation to tenancy models

The policy mechanism supports both siloed and pooled tenancy: - Siloed tenancy: Most domains default to `FROM_CREDENTIAL` (each tenant writes to its own partition). Only a few shared services (e.g., HR, catalog) use `FIXED` to centralize data. - Pooled tenancy: You may lean on `FIXED` policies more often to route writes into pooled/segment-specific domains (e.g., region, product line), while still enforcing read/write scoping via permissions.

Because the resolver always validates through the principal context and falls back safely, you can introduce overrides gradually without destabilizing existing flows.

63.9. Authoring tips

- Start with no policy and verify your default flows. Add entries only where necessary.
- Prefer specific keys (area:domain) for clarity; use wildcards sparingly.
- Keep `FIXED DataDomain` objects minimal and valid for your deployment (orgRefName, tenantId, and dataSegment as needed).
- Document any global policy so teams know which areas are centralized.

63.10. API pointers

- `CredentialUserIdPassword.dataDomainPolicy`: optional per-credential overrides (propagated to `PrincipalContext`).
- `GlobalDataDomainPolicyProvider`: holds an optional in-memory global policy (null by default).
- `DataDomainPolicyEntry.resolutionMode`: `FROM_CREDENTIAL` (default) or `FIXED`.
- `DataDomainResolver` / `DefaultDataDomainResolver`: the extension point and default behavior.
=== Security Annotations: `FunctionalMapping` and `FunctionalAction`

Quantum uses annotations to declare a model or resource's functional area, domain, and actions for security evaluation. This replaces the legacy `bmFunctionalArea()` and `bmFunctionalDomain()` methods.

63.10.1. @FunctionalMapping

Use `@FunctionalMapping` on model classes or resource classes to declare their business placement:

```
import com.e2eq.framework.annotations.FunctionalMapping;

@Entity
@FunctionalMapping(area = "catalog", domain = "product")
public class Product extends BaseModel {
    // No need to override bmFunctionalArea/bmFunctionalDomain
}
```

```

@Path("/products")
@FunctionalMapping(area = "catalog", domain = "product")
public class ProductResource extends BaseResource<Product, ProductRepo> {
    // All methods inherit area/domain from class annotation
}

```

63.10.2. @FunctionalAction

Use `@FunctionalAction` on JAX-RS resource methods when the action differs from the HTTP verb default:

```

@Path("/products")
public class ProductResource {

    @POST
    @FunctionalAction("CREATE") // Explicit, though POST implies CREATE
    public Product create(Product payload) {
        return productRepo.save(payload);
    }

    @GET
    @Path("/{id}")
    // No annotation needed - GET implies VIEW
    public Product get(@PathParam("id") String id) {
        return productRepo.findById(id);
    }

    @PUT
    @Path("/{id}/approve")
    @FunctionalAction("APPROVE") // Custom action beyond standard CRUD
    public Product approve(@PathParam("id") String id) {
        Product p = productRepo.findById(id);
        p.setStatus("APPROVED");
        return productRepo.save(p);
    }
}

```

63.10.3. Default Action Mapping

When `@FunctionalAction` is not present, actions are inferred from HTTP methods:

HTTP Method	Default Action
GET	VIEW
POST	CREATE
PUT	UPDATE
PATCH	UPDATE

HTTP Method	Default Action
DELETE	DELETE

63.10.4. How the Framework Uses These Annotations

SecurityFilter

- Reads `@FunctionalMapping` from the matched resource class for area/domain
- Reads `@FunctionalAction` from the method, or infers from HTTP method
- Falls back to path-based parsing if annotations are missing

MorphiaRepo.fillUIActions

- Uses `@FunctionalMapping` on model classes to resolve allowed UI actions
- Falls back to legacy `bmFunctionalArea()/bmFunctionalDomain()` methods

PermissionResource

- Prefers `@FunctionalMapping` when listing functional domains
- Falls back to legacy methods when annotation is missing

63.10.5. Migration from Legacy Methods

Current (Legacy) Approach

```
@Entity
public class Product extends BaseModel {
    @Override
    public String bmFunctionalArea() {
        return "Catalog";
    }

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

New (Recommended) Approach

```
@Entity
@FunctionalMapping(area = "catalog", domain = "product")
public class Product extends BaseModel {
    // Clean - no method overrides needed
}
```

Transitional Support

You can use both during migration: - If `@FunctionalMapping` is present, it takes precedence - If missing, legacy methods are used as fallback - Plan to remove legacy methods in future releases

63.10.6. Best Practices

Consistent Naming

Use lowercase, kebab-case for areas and domains:

```
@FunctionalMapping(area = "supply-chain", domain = "purchase-order")
@FunctionalMapping(area = "catalog", domain = "product")
@FunctionalMapping(area = "identity", domain = "user-profile")
```

Resource vs Model Annotations

- **Prefer model annotations** for consistency across all usage
- Use resource annotations only when the resource handles multiple model types
- Avoid duplicating annotations on both model and resource for the same entity

Custom Actions

Define custom actions for business operations beyond CRUD:

```
@PUT
@Path("/{id}/publish")
@FunctionalAction("PUBLISH")
public Product publish(@PathParam("id") String id) { ... }

@POST
@Path("/{id}/duplicate")
@FunctionalAction("DUPLICATE")
public Product duplicate(@PathParam("id") String id) { ... }

@DELETE
@Path("/{id}/archive")
@FunctionalAction("ARCHIVE") // Soft delete vs hard DELETE
public void archive(@PathParam("id") String id) { ... }
```

63.10.7. Integration with Permission Rules

Annotations feed into permission rule matching:

```
- name: allow-catalog-reads
  priority: 300
  match:
```

```
method: [GET]
# Matches area/domain from @FunctionalMapping
functionalArea: catalog
functionalDomain: product
rolesAny: [USER, ADMIN]
effect: ALLOW
```

```
- name: admin-only-approval
priority: 100
match:
  method: [PUT]
  functionalArea: catalog
  functionalDomain: product
  # Matches @FunctionalAction("APPROVE")
  action: APPROVE
rolesAll: [ADMIN]
effect: ALLOW
```

63.10.8. See Also

- [Modeling: Functional Mapping](#)
- [Permission Rules](#)
- [DomainContext and RuleContext](#)

Chapter 64. 9. Seed packs: Declarative tenant seeding

Problem: Provisioning repeatable, versioned baseline data for tenants.

Why for SaaS: Every tenant must start with known-good defaults; upgrades must be idempotent and auditable.

How Quantum helps: Seed packs with manifests, datasets, transforms, includes, archetypes, and a registry.

Walkthrough: Create a seed pack, apply it programmatically, and verify idempotency.

Seed packs and declarative tenant seeding

Quantum 1.2 introduces a seed-pack subsystem that lets applications publish versioned baseline content without hard-coding values in **ChangeSet** beans or maintaining a separate "seed" tenant.

Chapter 65. Introduction

65.1. The problem

In multi-tenant SaaS platforms, every new tenant must start with a known-good baseline of data: code lists, roles, default settings, reference values, and sometimes product- or region-specific content. Traditionally this baseline is scattered across ad-hoc SQL/Mongo scripts, hand-written bootstrap code, or a "template" tenant that is copied forward. These approaches are hard to version, review, test, and repeat reliably across environments.

Compounding the issue, tenants evolve over time. As modules are upgraded, their baseline content must be updated too. Without a disciplined mechanism, teams risk drift between environments and tenants, brittle migrations, and non-idempotent provisioning that causes duplicates or corruption.

65.2. Why this needs to be solved

- Operational consistency: Provisioning should be predictable, repeatable, and safe to re-run.
- Developer velocity: Changes to baseline data should be reviewed like code and travel with the module that owns them.
- Compliance and audit: You need to know exactly which version of seed content was applied to which tenant and when.
- Composability: Different product editions or SKUs need different combinations of baseline content without forked scripts.

65.3. How seed packs solve it

Seed packs provide a declarative, versioned, and composable way to describe tenant baseline data:

- A manifest (manifest.yaml) declares datasets, natural keys, transforms, required indexes, and optional includes/archetypes.
- Datasets point to JSON/NDJSON files that are upserted using natural-key filters, making runs idempotent.
- Transforms inject tenant/realm identifiers and can rewrite references deterministically.
- Includes compose other packs with exact versions or semantic version ranges, enabling dependency management.
- Archetypes bundle a named set of packs to represent product tiers or verticals.
- A registry records checksums per dataset so unchanged data is skipped on subsequent runs.

Together, these features make seeding safe, observable, and maintainable across development, test, and production.

The next section expands on why seed packs are beneficial and how to use them effectively.

Chapter 66. Why seed packs?

- **Versioned + reviewable:** seed packs are plain files (YAML + JSON/NDJSON) that live next to your module code. Pull requests show exactly which records changed.
- **Composable:** packs can depend on other packs and expose named *archetypes* for different product editions or verticals.
- **Pluggable sources:** load packs from the filesystem, object storage, or even a curated seed database by providing a custom **SeedSource**.
- **Tenant-aware:** transforms inject tenant identifiers and remap references before persisting.
- **Idempotent:** a **SeedRegistry** tracks checksums per dataset so provisioning can be re-run safely.

Chapter 67. High-level flow

1. `SeedLoader` discovers manifests via the configured `SeedSource` implementations (for example the provided `FileSeedSource`).
2. A manifest (`manifest.yaml`) declares datasets, required indexes, transforms, optional includes, and archetypes.
3. During provisioning a migration invokes `SeedLoader.apply(...)` with the packs (or archetype) that should be materialised for the tenant.
4. Records are parsed, transformed, and upserted through a `SeedRepository` implementation. The default `MongoSeedRepository` writes to the tenant realm using natural-key filters.
5. The `SeedRegistry` (backed by `_seed_registry` via `MongoSeedRegistry`) records the checksum so unchanged datasets are skipped on later runs.

Chapter 68. Manifest quick reference

```
seedPack: logistics-core
version: 1.4.2
includes:
  - accounting-base@^1.1

datasets:
  - collection: codeLists
    file: datasets/codelists.ndjson
    naturalKey: [codeListName, code]
    upsert: true
    requiredIndexes:
      - name: uk_codeLists_name_code
        unique: true
        keys:
          codeListName: 1
          code: 1
    transforms:
      - type: tenantSubstitution
        config:
          tenantField: tenantId
          orgField: orgId
          ownerField: ownerId
          realmField: realmId

archetypes:
  - name: FulfillmentPlus
    includes:
      - logistics-core@^1.4
      - shipping-defaults@~2
```

Chapter 69. Programmatic usage

```
SeedLoader loader = SeedLoader.builder()
    .addSeedSource(new FileSeedSource("local", Paths.get("seed-packs")))
    .seedRepository(new MongoSeedRepository(mongoClient))
    .seedRegistry(new MongoSeedRegistry(mongoClient))
    .build();

SeedContext ctx = SeedContext.builder(realmId)
    .tenantId(tenantId)
    .orgRefName(orgRef)
    .accountId(accountId)
    .ownerId(ownerId)
    .build();

loader.apply(List.of(
    SeedPackRef.range("logistics-core", "^1.4"),
    SeedPackRef.of("oms-defaults")
), ctx);
```

Callers can also use `loader.applyArchetype("FulfillmentPlus", ctx)` to resolve an archetype defined in any manifest.

Chapter 70. Extensibility hooks

- Implement `SeedSource` to load manifests from custom storage (S3, Git, curated seed DB...).
- Register additional `SeedTransformFactory` instances with the builder to support bespoke transformations (for example JMESPath projections or deterministic ObjectId mapping).
- Swap in a different `SeedRepository/SeedRegistry` to write to alternative datastores or change the idempotency policy.

Chapter 71. Operational tips

- Validate manifests in CI by running the loader against a disposable database.
- Keep seed pack versions aligned with module versions so upgrade paths are clear.
- Derive any ObjectIds deterministically from natural keys inside a transform so data can be re-applied without collisions.
- Use archetypes to model product tiers and optional modules: `TenantProvisioningService` can decide which archetype(s) to apply based on SKU.

Chapter 72. Primary scenarios

1. Initial tenant provisioning

- Apply one or more seed packs to bootstrap a brand-new tenant (realm) with baseline code lists, roles, and default settings.
- Use `SeedPackRef.of("pack-name")` or `SeedPackRef.range("pack-name", "^1.4")` to control versions.

2. Updating a module to a new version

- Publish a new seed pack version (e.g., logistics-core 1.5.0) with incremental dataset changes.
- Re-run `loader.apply(...)` for the same tenant; unchanged datasets are skipped via `_seed_registry`, modified datasets are re-applied.

3. Idempotent re-apply during deployments

- Safe to invoke on every startup/migration. Upserts are driven by `naturalKey` and `upsert: true`.
- Keep natural keys stable; derive surrogate IDs deterministically in a transform if needed.

4. Selecting product tiers with archetypes

- Define archetypes in a manifest to bundle multiple seed packs under a named edition.
- Call `loader.applyArchetype("FulfillmentPlus", ctx)` to materialize the predefined stack for a tenant.

5. Composing packs with includes

- Use includes to depend on base packs (e.g., accounting-base@^1.1) and extend with your own datasets.
- Includes support `exact (=1.2.3)` and `range` (e.g., `^1.4`, `~2`) selectors via `SeedPackRef.parse("name@spec")`.

6. Partial refresh of specific datasets

- You can split large packs into multiple datasets and re-apply only the packs you want by passing a smaller list to `loader.apply(...)`.

7. Testing seed packs

- Add an integration test similar to `SeedLoaderIntegrationTest` that seeds into an ephemeral MongoDB and asserts collection state and `_seed_registry` entries.

Chapter 73. Explicit examples

73.1. Example 1: Minimal manifest and NDJSON

```
seedPack: demo-seed
version: 1.0.0

datasets:
- collection: codeLists
  file: datasets/codeLists.ndjson
  naturalKey: [ code ]
  upsert: true
  requiredIndexes:
  - name: uk_codeLists_code
    unique: true
    keys:
      code: 1
  transforms:
  - type: tenantSubstitution
    config:
      tenantField: tenantId
      orgField: orgRefName
      accountField: accountId
      ownerField: ownerId
      realmField: realmId
```

Example NDJSON (datasets/codeLists.ndjson):

```
{"code": "NEW", "label": "New"}
{"code": "CLOSED", "label": "Closed"}
```

73.2. Example 2: Applying packs in code

```
SeedLoader loader = SeedLoader.builder()
    .addSeedSource(new FileSeedSource("local", Paths.get("seed-packs")))
    .seedRepository(new MongoSeedRepository(mongoClient))
    .seedRegistry(new MongoSeedRegistry(mongoClient))
    .build();

SeedContext ctx = SeedContext.builder("my-realm")
    .tenantId("tenant-123")
    .orgRefName("tenant-123")
    .accountId("acct-123")
    .ownerId("owner-123")
    .build();
```

```
loader.apply(List.of(
    SeedPackRef.of("demo-seed"),
    SeedPackRef.range("logistics-core", "^1.4")
), ctx);
```

73.3. Example 3: Using an archetype

```
archetypes:
- name: FulfillmentPlus
  includes:
  - logistics-core@^1.4
  - shipping-defaults@~2
```

Apply programmatically:

```
loader.applyArchetype("FulfillmentPlus", ctx);
```

73.4. Example 4: Exact version and includes in a manifest

```
seedPack: shipping-defaults
version: 2.3.0
includes:
- accounting-base@=1.1.2
- logistics-core@^1.5

datasets:
- collection: shippingMethods
  file: datasets/methods.json
  naturalKey: [ code ]
```

Chapter 74. Troubleshooting

- Manifest parsing errors: Confirm manifest.yaml keys match SeedPackManifest fields; boolean flags like upsert and unique must be proper booleans.
- Duplicate key or unique index violations: Check naturalKey and requiredIndexes; ensure transforms don't change key fields inconsistently.
- Nothing changes on re-run: The _seed_registry may have recorded the same checksum; bump version or change dataset content.
- File resolution issues: Ensure FileSeedSource base path points to the correct seed-packs directory and file names match.

Chapter 75. How seeds are applied automatically at startup

The framework now applies seed packs via a dedicated `SeedStartupRunner`, independent of schema migrations. This runner discovers and applies the latest version of each seed pack for important realms (system/default/test) on application startup.

Key points: - Discovery: `SeedStartupRunner` constructs a `SeedLoader` with a `FileSeedSource` pointing at the configured seed root. Set `quantum.seed.root` (for tests we default to `src/test/resources/seed-packs`). The source walks the directory tree and locates every `manifest.yaml` file. - Selection: For each discovered seed pack name, the runner selects the latest semantic version and builds `SeedPackRef.exact(name, version)` for application. - Execution: The runner builds a `SeedContext` for the target realm and calls `loader.apply(refs, context)`. Indexes declared in the manifest are created before data is upserted. - Idempotency + repeatable: The `MongoSeedRegistry` stores a checksum per dataset in the realm's `_seed_registry` collection. If the checksum matches on a later run, the dataset is skipped; if it changes, the dataset is re-applied. - Concurrency safety: The runner uses a Sherlock distributed lock per realm to prevent concurrent execution across nodes.

Configuration snippet:

```
# test profile uses a local seed root
quantum.seed.root=src/test/resources/seed-packs
# control seed runner behavior
quantum.seeds.enabled=true
quantum.seeds.apply.on-startup=true
```

Chapter 76. Transforms in depth

Transforms are small, composable functions that shape each dataset record just before it is written to the database. They let you keep dataset files generic and inject environment/tenant specifics or perform repeatable rewrites at apply time.

What a transform gets and returns: - Input: the current record (a Map), the SeedContext, and the Dataset definition - Output: the next record (Map) to be passed to the rest of the pipeline; return null or an empty map to drop the record

Where transforms are declared (manifest):

```
datasets:
- collection: codeLists
  file: datasets/codeLists.ndjson
  naturalKey: [ codeListName, code ]
  upsert: true
  transforms:
    - type: tenantSubstitution
      config:
        tenantField: tenantId
        orgField: orgRefName
        ownerField: ownerId
        accountField: accountId
        realmField: realmId
    # Additional transforms can be added here and will execute in order
```

Execution semantics: - Ordering: transforms are executed top-to-bottom for each record - Short-circuit: if any transform returns null or an empty map, the record is skipped and no write occurs - Overwrite rules: a transform can set or overwrite fields on the record; when upsert=true, the final transformed record replaces the existing one matched by naturalKey - Interaction with naturalKey and indexes: transforms run before naturalKey validation and index creation; do not remove fields listed in naturalKey, otherwise an error will be thrown during write

Built-in transform types: - tenantSubstitution: Injects identifiers from SeedContext into configured fields. Config keys in the manifest: - tenantField: field in the record to receive tenantId (if present in context) - orgField: field to receive orgRefName (if present) - ownerField: field to receive ownerId (if present) - accountField: field to receive accountId (if present) - realmField: field to receive realmId (always set to the current realm)

Notes and guarantees: - Optional values missing from SeedContext are simply omitted; existing record values are preserved unless you target the same field - Transforms operate on in-memory maps and cannot perform I/O by default; keep them deterministic so re-runs are idempotent - Compose multiple transforms when needed (for example: first tenantSubstitution, then a custom id computation) - To add new transform types, implement SeedTransformFactory and register it during SeedLoader.builder() with registerTransformFactory("myType", new MyFactory()); then declare - type: myType in the manifest

When to use transforms (and why): - Injecting tenant/realm identity: keep datasets source-controlled and generic; inject tenant IDs at apply time (tenantSubstitution) - Deterministic IDs: derive `_id` or other surrogate keys from naturalKey so upserts remain stable across environments - Normalization and defaults: add missing fields, convert formats, enforce enums before write - Reference remapping: translate human-readable codes in the dataset into datastore-specific identifiers (ObjectIds, UUIDs) in a repeatable way

Practical examples

1) Tenant identity injection (built-in) Manifest snippet:

```
transforms:
- type: tenantSubstitution
  config:
    tenantField: tenantId
    orgField: orgRefName
    ownerField: ownerId
    accountField: accountId
    realmField: realmId
```

Why: keep `codeLists.ndjson` portable across tenants; provisioning injects the right IDs based on `SeedContext`.

2) Deterministic `_id` from natural key (custom) - Goal: ensure stable MongoDB `_id` across re-applies and environments, derived from `codeListName+code` - Approach: implement a custom transform that computes a SHA-1/MD5 hash (or any deterministic function) and sets `_id`

Java registration:

```
SeedLoader loader = SeedLoader.builder()
    .addSeedSource(new FileSeedSource("local", Paths.get("seed-packs")))
    .seedRepository(new MongoSeedRepository(mongoClient))
    .seedRegistry(new MongoSeedRegistry(mongoClient))
    .registerTransformFactory("deterministicId", new DeterministicIdTransform.Factory())
    .build();
```

Manifest usage:

```
transforms:
- type: tenantSubstitution
- type: deterministicId
  config:
    sourceFields: [ codeListName, code ]
    targetField: _id
    algorithm: sha1
```

Why: makes upserts resilient and allows cross-environment joins by a stable key.

3) Foreign key remapping by code (custom) - Goal: dataset uses human-readable statusCode; transform maps it to a canonical statusId - Approach: custom transform with an in-memory map or deterministic derivation

Manifest usage:

```
transforms:
  - type: mapCode
    config:
      field: statusCode
      target: statusId
      mapping:
        NEW: 100
        CLOSED: 900
```

Why: keeps datasets human-friendly while persisting efficient identifiers.

4) Defaulting and sanitization (custom) - Goal: ensure missing fields get defaults and strings are trimmed/lowercased - Approach: simple custom transform that fills defaults and cleans values

Manifest usage:

```
transforms:
  - type: defaults
    config:
      defaults:
        isActive: true
        locale: en_US
  - type: sanitize
    config:
      trim: [ label ]
      lowercase: [ email ]
```

Why: enforces consistency without editing large datasets.

Testing transforms: - Add integration tests that seed into an ephemeral DB and assert both record shape and `_seed_registry` entries - For custom transforms, add focused unit tests for edge cases (missing fields, nulls, unexpected types)

76.1. Creating your own transforms (example: DropIfTransform)

Custom transforms let you implement project-specific shaping logic. You implement two small interfaces and register the type on the SeedLoader builder. Below we walk through a simple "drop the record if a field equals a value" transform used in tests, called DropIfTransform.

Overview of the SPI: - SeedTransform: executes per-record and can return a new map (continue) or null/empty (drop this record). - SeedTransformFactory: builds a SeedTransform instance from the manifest's Transform definition (provides access to type and config map).

Minimal interfaces (simplified for clarity):

```
public interface SeedTransform {
    Map<String, Object> apply(Map<String, Object> record,
                             SeedContext context,
                             SeedPackManifest.Dataset dataset);
}

public interface SeedTransformFactory {
    SeedTransform create(SeedPackManifest.Transform transformDefinition);
}
```

Implementation: DropIfTransform - Behavior: if record[field] equals a configured value, return null to short-circuit the pipeline and skip writing the record; otherwise, pass the record through unchanged.

```
package com.example.seed.transforms;

import com.e2eq.framework.service.seed.*;
import java.util.Map;
import java.util.Objects;

public final class DropIfTransform implements SeedTransform {
    private final String field;
    private final String equalsValue;

    public DropIfTransform(String field, String equalsValue) {
        this.field = field;
        this.equalsValue = equalsValue;
    }

    @Override
    public Map<String, Object> apply(Map<String, Object> record, SeedContext context,
    SeedPackManifest.Dataset dataset) {
        Object v = record.get(field);
        if (Objects.equals(Objects.toString(v, null), equalsValue)) {
            return null; // short-circuit: drop this record
        }
        return record;
    }

    public static final class Factory implements SeedTransformFactory {
        @Override
        public SeedTransform create(SeedPackManifest.Transform transformDefinition) {
            Map<String, Object> cfg = transformDefinition.getConfig();
        }
    }
}
```



```

    String field = Objects.toString(cfg.get("field"), null);
    String eq = Objects.toString(cfg.get("equals"), null);
    return new DropIfTransform(field, eq);
  }
}

```

Registration on the SeedLoader builder:

```

SeedLoader loader = SeedLoader.builder()
    .addSeedSource(new FileSeedSource("local", Paths.get("seed-packs")))
    .seedRepository(new MongoSeedRepository(mongoClient))
    .seedRegistry(new MongoSeedRegistry(mongoClient))
    .registerTransformFactory("dropIf", new DropIfTransform.Factory())
    .build();

```

Manifest usage:

```

datasets:
- collection: codeLists
  file: datasets/codeLists.ndjson
  naturalKey: [ code ]
  upsert: true
  transforms:
  - type: dropIf
    config:
      field: status
      equals: CLOSED

```

Notes and tips: - Validation: your factory should validate required config keys and fail fast with a clear error if missing/invalid. - Determinism: keep transforms pure and deterministic (no I/O) so seeding remains idempotent. - Short-circuit: returning null or an empty map drops the record; otherwise, the next transform in the list will receive the (possibly mutated) map. - Composition: you can chain several transforms; for example, first dropIf, then tenantSubstitution, then a custom deterministicId. - Packaging: test-only transforms can live under test sources; production transforms should be in main sources and registered where you construct the SeedLoader (for example, in a ChangeSet or a provisioning service).

Chapter 77. Test walkthrough:

SeedLoaderIntegrationTest

The `SeedLoaderIntegrationTest` demonstrates end-to-end seeding using the demo seed pack at `src/test/resources/seed-packs/demo-seed`.

What the test does:

- Creates a `SeedLoader` backed by `FileSeedSource`, `MongoSeedRepository`, and `MongoSeedRegistry`.
- Builds a `SeedContext` populated with tenant/realm details to exercise the `tenantSubstitution` transform.
- Applies the pack reference `SeedPackRef.of("demo-seed")`, which resolves the latest version of that pack (1.0.0 in tests).
- Asserts that 2 records were inserted into the `codeLists` collection and that the `tenantSubstitution` fields were populated from the context.
- Verifies an entry was written to `_seed_registry` with the dataset checksum and `records: 2`.
- Re-applies the same pack and asserts the record count remains 2, demonstrating idempotency (thanks to upsert + registry checksum).

Why these design choices:

- NDJSON for datasets: allows streaming large datasets and simple line-by-line diffs in code review; arrays are also supported for smaller payloads.
- Natural-key upsert: manifests declare `naturalKey` to form the filter for `replaceOne(..., upsert=true)` ensuring idempotent writes and predictable overwrites.
- Transform pipeline: keeps dataset files free of environment-specific values; all tenant/realm specifics are injected consistently at apply time.
- Registry-based skip: checksums per dataset avoid unnecessary writes when content hasn't changed—fast, safe re-runs during deployments.
- Semantic-version selection: when multiple versions of a pack are available, the latest semver is used unless an exact version is requested.

Alternatives considered:

- Store seed state in an external table keyed only by version. Rejected in favor of per-dataset checksums to detect content drift without bumping versions.
- Hardcode seeding logic inside ad-hoc migrations. Rejected for lack of composability and poor reviewability.
- Use inserts only (no upsert). Rejected due to lack of idempotence and difficulty correcting baseline data.

Chapter 78. Archetypes explained

Archetypes are named compositions of seed packs that model a product edition, SKU, or vertical stack. Instead of listing several packs every time you provision a tenant, you define an archetype once in a manifest and then apply it by name.

What an archetype is in this context: - It lives inside a seed pack manifest under `archetypes:`. - It contains a list of includes (same syntax as top-level includes) referring to packs and version ranges. - When applied, the loader resolves those pack refs plus the hosting pack itself (the manifest that defines the archetype) so that local datasets are included as part of the archetype. - Resolution uses semantic version rules and deduplicates by pack name, respecting dependency order and preventing cycles.

When to use archetypes: - To represent product tiers (e.g., Community, Pro, Enterprise) that bundle different combinations of base packs and optional modules. - To group verticalized defaults (e.g., Logistics-Fulfillment, Healthcare-Core) without forcing consumers to know every underlying pack.

78.1. Example A: Define and apply an archetype in the same pack

Manifest (logistics-core/manifest.yaml):

```
seedPack: logistics-core
version: 1.4.2

includes:
  - accounting-base@^1.1

datasets:
  - collection: codeLists
    file: datasets/codeLists.ndjson
    naturalKey: [ codeListName, code ]
    upsert: true

archetypes:
  - name: FulfillmentPlus
    includes:
      - logistics-core@^1.4      # self + constraints
      - shipping-defaults@~2
```

Applying it in code:

```
SeedLoader loader = SeedLoader.builder()
    .addSeedSource(new FileSeedSource("local", Paths.get("seed-packs")))
    .seedRepository(new MongoSeedRepository(mongoClient))
    .seedRegistry(new MongoSeedRegistry(mongoClient))
    .build();
```

```
SeedContext ctx = SeedContext.builder("my-realm").build();
loader.applyArchetype("FulfillmentPlus", ctx);
```

Notes: - `applyArchetype` looks up the latest manifest that defines an archetype named "FulfillmentPlus" across all discovered packs, resolves the include refs, and then applies the union. - If multiple manifests define the same archetype name, the latest semver manifest wins.

78.2. Example B: Cross-pack archetype in a dedicated "editions" pack

You can centralize product definitions into a thin pack that only defines archetypes and forward-references other packs:

Manifest (product-editions/manifest.yaml):

```
seedPack: product-editions
version: 1.0.0

archetypes:
- name: Enterprise
  includes:
    - logistics-core@^1.5
    - shipping-defaults@~2
    - analytics-starter@^0.9
```

Apply in code:

```
loader.applyArchetype("Enterprise", ctx);
```

This keeps edition composition decoupled from individual module packs.

78.3. Resolution and ordering details

- Version matching: Each include can be exact (=1.2.3), a semver range (e.g., ^1.5, ~2), or omitted (latest). See `SeedPackRef.parse("name@spec")`.
- Deduplication: If multiple includes select the same pack name (possibly different versions), the highest version that satisfies all constraints is chosen; duplicates are applied only once.
- Dependency order: Includes are recursively resolved depth-first, while the loader guards against cycles and applies datasets in a stable order per resolved pack.

78.4. Interaction with `ApplySeedPacksChangeSet`

- The `Apply Seed Packs` change set scans the seed root and applies the latest version of every

discovered pack to the realm. It does not automatically choose an archetype.

- Use `applyArchetype` programmatically (e.g., from a `TenantProvisioningService`) when you want to provision only the packs that belong to a specific edition.
- You can combine approaches: let migrations ensure baseline packs are present for all tenants; then, on tenant onboarding, call `applyArchetype(...)` to add edition-specific content.

78.5. Tenant provisioning with archetypes

The tenant provisioning API accepts an optional list of archetype names and will apply the corresponding seed packs during onboarding.

- Endpoint: `POST /admin/tenants`
- Request body fields (subset):
- `tenantEmailDomain`, `orgRefName`, `accountId`, `adminUserId`, `adminUsername`, `adminPassword`
- `archetypes`: optional array of strings (archetype names)
- Behavior:
- After running migrations and creating the admin user, each archetype is resolved across all manifests and applied using the same `SeedLoader` used elsewhere.
- If an archetype name is unknown, the request fails with 409/500 depending on context.

Example request:

```
{
  "tenantEmailDomain": "demo-archetype.example",
  "orgRefName": "demo-archetype.example",
  "accountId": "999999999",
  "adminUserId": "admin@demo-archetype.example",
  "adminUsername": "admin@demo-archetype.example",
  "adminPassword": "secret",
  "archetypes": ["DemoArchetype"]
}
```

On success, the new realm (demo-archetype-example) will have the datasets from the selected archetypes applied and recorded in the `_seed_registry`.

Chapter 79. REST API for seed packs

The framework exposes admin-only endpoints to inspect and apply seed packs per realm (tenant DB). These are disabled to non-admin users via role checks.

Base path: /admin/seeds

Endpoints: - GET /admin/seeds/pending/{realm} - Lists pending seed packs for the realm. A pack is pending if any dataset checksum differs from the last applied or was never applied. - Optional query parameter: filter=pack1,pack2 to restrict by pack name. - Response example:

+

```
[
  {
    "seedId": "demo-seed@1.0.0",
    "seedPack": "demo-seed",
    "version": "1.0.0",
    "datasets": [
      { "collection": "codeLists", "file": "datasets/codelists.ndjson", "checksum":
"..."}
    ]
  }
]
```

- POST /admin/seeds/apply/{realm}
- Applies the latest version of all discovered packs (or only those matching filter).
- Query parameter: filter=pack1,pack2
- Response example:

```
{"applied":["demo-seed"]}
```

- POST /admin/seeds/{realm}/{seedPack}/apply
- Applies the latest version of a single pack by name. Idempotent: unchanged datasets are skipped.
- Response example:

```
{"applied":["demo-seed"]}
```

- GET /admin/seeds/history/{realm}
- Returns the per-dataset seeding history as recorded in the `_seed_registry` collection.

Authentication and roles: - All endpoints require role admin. In integration tests you can use `@TestSecurity(user="admin", roles={"admin"})`.

Configuration: - quantum.seed.root: filesystem path to the root folder where seed packs are discovered. In tests this defaults to src/test/resources/seed-packs. - quantum.seed.apply.filter: optional comma-separated list of pack names to limit automatic application by changeset.

Chapter 80. 10. Migrations

Problem: Evolving schemas and data safely across environments.

Why for SaaS: Continuous delivery means frequent, incremental change with strict uptime and audit needs.

How Quantum helps: Opinionated patterns and hooks for applying safe migrations.

Walkthrough: Add and run a simple migration.

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 81. 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 82. 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 $(\text{major} \times 100) + (\text{minor} \times 10) + \text{patch}$ (e.g., 1.0.3 \rightarrow 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 83. 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 84. 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 85. 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 86. 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 87. 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 88. 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 89. 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 90. 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 \geq required version; helpful for preflight checks.

Chapter 91. 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.

Chapter 92. 11. Testing

Problem: Validating behavior quickly and reliably, including tenant-specific behavior.

Why for SaaS: Isolation, entitlements, and data variance require strong test coverage.

How Quantum helps: Test utilities, runtime hooks, and integration testing guidance.

Walkthrough: Write integration tests for APIs and seed packs.

Chapter 93. Testing in Quantum: Security Contexts, Repos, and REST APIs

93.1. Testing Framework

This guide explains how to write effective tests in the Quantum framework, with a focus on setting the security context and choosing the right Quarkus testing patterns for repository logic vs. REST APIs.

It covers four practical patterns you can use to establish the security context in tests: - Extending `BaseRepoTest` - Using a try-with-resources `SecuritySession` - Using a scoped-call pattern to wrap work under a `SecuritySession` - Using the `@TestSecurity` annotation

In addition, it outlines how to test REST endpoints vs. repository logic and highlights useful features of the Quarkus Test Framework.

93.2. Prerequisites and glossary

- `RuleContext`: the in-memory rule engine configuration used by authorization checks.
- `PrincipalContext` (`pContext`): who is performing the action (`userId`, roles, realms, data domain).
- `ResourceContext` (`rContext`): what is being acted upon (area/domain, resource, action).
- `SecuritySession`: a small utility that binds `PrincipalContext` and `ResourceContext` to `SecurityContext` for the current thread, and clears them on close.
- `SecurityIdentity`: Quarkus' current identity (used by `@TestSecurity` and HTTP request security).

93.3. Pattern 1 — Extend `BaseRepoTest`

For repository tests (no HTTP), the simplest approach is to extend `BaseRepoTest`.

What `BaseRepoTest` does for you: - Initializes `RuleContext` with default rules. - Builds default `pContext` and `rContext` for a system/test user. - Ensures test database migrations are applied. - Gives you protected fields `pContext` and `rContext` you can use in your test.

Example:

```
@QuarkusTest
class MyRepoTest extends com.e2eq.framework.persistent.BaseRepoTest {

    @Inject
    com.e2eq.framework.model.persistent.morphia.UserProfileRepo userProfileRepo;

    @Test
    void canReadUnderTestUser() {
        // Activate the security context for this block
        try (final com.e2eq.framework.securityrules.SecuritySession ignored =
```

```

        new com.e2eq.framework.securityrules.SecuritySession(pContext, rContext))
    {
        var list = userProfileRepo.list(testUtils.getTestRealm());
        org.junit.jupiter.api.Assertions.assertNotNull(list);
    }
}
}

```

Notes: - BaseRepoTest prepares contexts but does not keep them permanently active. Wrap repo calls that require authorization in a SecuritySession (see Pattern 2), or activate it in @BeforeEach if many test methods use it. - BaseRepoTest also runs migrations once using your test principal, which avoids authorization failures during initialization.

93.4. Pattern 2 — Try-with-resources SecuritySession

Use SecuritySession explicitly to scope work that should run under a specific PrincipalContext and ResourceContext. This is the most explicit pattern and works in both repository and service-level tests.

Example:

```

@QuarkusTest
class CredentialRepoTest extends com.e2eq.framework.persistent.BaseRepoTest {

    @Inject
    com.e2eq.framework.model.persistent.morphia.CredentialRepo credRepo;

    @Test
    void findById_asTestUser() {
        try (final com.e2eq.framework.securityrules.SecuritySession s =
            new com.e2eq.framework.securityrules.SecuritySession(pContext, rContext))
        {
            var op = credRepo.findById(testUtils.getTestUserId(), testUtils
                .getSystemRealm());
            org.junit.jupiter.api.Assertions.assertTrue(op.isPresent());
        }
    }
}

```

Tips: - Prefer try-with-resources so contexts are always cleared, even when assertions fail. - You can construct custom PrincipalContext/ResourceContext for specific scenarios (e.g., different roles or realms) and pass them to SecuritySession.

93.5. Pattern 3 — Scoped-call wrapper (ScopedCallScope)

If you prefer not to repeat try-with-resources blocks, wrap your work in a helper that creates a

SecuritySession, runs your logic, and ensures cleanup. This is sometimes called a “scoped call” pattern, often referred to as a ScopedCallScope.

Example helper (placed in test sources):

```
public final class SecurityScopes {
    private SecurityScopes() {}

    public static <T> T call(
        com.e2eq.framework.model.securityrules.PrincipalContext p,
        com.e2eq.framework.model.securityrules.ResourceContext r,
        java.util.concurrent.Callable<T> work) {
        try (final com.e2eq.framework.securityrules.SecuritySession s =
            new com.e2eq.framework.securityrules.SecuritySession(p, r)) {
            try { return work.call(); }
            catch (Exception e) { throw new RuntimeException(e); }
        }
    }

    public static void run(
        com.e2eq.framework.model.securityrules.PrincipalContext p,
        com.e2eq.framework.model.securityrules.ResourceContext r,
        Runnable work) {
        try (final com.e2eq.framework.securityrules.SecuritySession s =
            new com.e2eq.framework.securityrules.SecuritySession(p, r)) {
            work.run();
        }
    }
}
```

Usage:

```
var result = SecurityScopes.call(pContext, rContext, () -> repo.getByUserId(realm,
userId));
SecurityScopes.run(pContext, rContext, () -> repo.save(realm, entity));
```

This achieves the same effect as try-with-resources but centralizes the pattern.

93.6. Pattern 4 — @TestSecurity annotation (Quarkus)

For tests that run through HTTP (and in some repo tests), you can use Quarkus’ @io.quarkus.test.security.TestSecurity to set the SecurityIdentity without creating a SecuritySession.

Quantum integrates with this in two places: - SecurityFilter: when a request has a SecurityIdentity but no JWT, it builds a PrincipalContext from the identity (user and roles). You can also pass X-Realm to control the realm. - MorphiaRepo: when repo methods are invoked under @TestSecurity with no active SecuritySession, MorphiaRepo lazily builds PrincipalContext from SecurityIdentity and sets a safe default ResourceContext to enable rule evaluation.

Example (HTTP):

```
@QuarkusTest
class SecureResourceTest {

    @Inject com.e2eq.framework.util.TestUtils testUtils;

    @Test
    @io.quarkus.test.security.TestSecurity(user = "test@system.com", roles = {"user"})
    void listProfiles_asUser() {
        io.restassured.RestAssured.given()
            .header("X-Realm", testUtils.getTestRealm())
            .when().get("/user/userProfile/list")
            .then().statusCode(200);
    }
}
```

Example (repo call under @TestSecurity fallback, no SecuritySession):

```
@QuarkusTest
class RepoFallbackTest {

    @Inject com.e2eq.framework.util.TestUtils testUtils;
    @Inject com.e2eq.framework.model.persistent.morphia.CredentialRepo credentialRepo;

    @Test
    @io.quarkus.test.security.TestSecurity(user = "test@system.com", roles = {"user"})
    void repoUsesIdentityWhenNoSecuritySession() {
        // Internally, MorphiaRepo will ensure PrincipalContext exists using
        // SecurityIdentity
        credentialRepo.findById("nonexistent@end2endlogic.com", testUtils.
            getTestRealm(), false);
        // Optionally assert that SecurityContext has been initialized
        org.junit.jupiter.api.Assertions.assertTrue(
            com.e2eq.framework.model.securityrules.SecurityContext.getPrincipalContext()
                .isPresent());
    }
}
```

Notes: - @TestSecurity is perfect for authorizing requests in HTTP tests without generating JWTs. - For repo tests that require precise ResourceContext (area/domain/action), prefer SecuritySession; MorphiaRepo sets a generic default ResourceContext when needed.

93.7. Testing REST APIs vs. Repository Logic

When to prefer REST (HTTP) tests: - End-to-end authorization: validate request filters, identity mapping, realm headers, and JWT handling. - Request/response shape and status codes. - Role-based access checks via @TestSecurity.

How to test REST APIs: - Use `@QuarkusTest` and `RestAssured`: [source,java] ---- `var resp = io.restassured.RestAssured.given().header("Content-Type", "application/json").header("X-Realm", testUtils.getTestRealm()).when().get("/user/userProfile/list").then().statusCode(200).extract().response();` ---- - To test JWT-protected endpoints end-to-end, first call the login API to obtain a token, then pass `Authorization: Bearer <token>`. See `SecurityTest.testGetUserProfileRESTAPI` for a complete example.

When to prefer repository/service tests: - You want precise control over `PrincipalContext/ResourceContext` and rule evaluation without HTTP overhead. - You are asserting persistence logic, query filters, or domain rules.

How to test repository logic: - Extend `BaseRepoTest` (Pattern 1) for ready-to-use `pContext/rContext` and migrations. - Wrap calls with `SecuritySession` (Pattern 2) or use a scoped-call helper (Pattern 3).

93.8. Useful Quarkus Test features

- `@QuarkusTest`: boots the app for integration tests with CDI, config, and persistence.
- `RestAssured`: fluent HTTP client baked into Quarkus tests; supports JSON assertions and extraction.
- `@TestSecurity`: set `SecurityIdentity` (user, roles) for tests.
- `@InjectMock/@InjectSpy` (quarkus-junit5-mockito): replace beans with mocks/spies for isolation.
- `@QuarkusTestResource`: manage external resources (e.g., starting/stopping containers) for a test class or suite.
- `@TestHTTPEndpoint` and `@TestHTTPResource`: convenient endpoint URI injection.

93.9. Real-world tips

- Clearing thread locals: If you manipulate `SecurityContext` directly in advanced tests, clear it in `@AfterEach` to avoid cross-test leakage: [source,java] ---- `@AfterEach void cleanup() { com.e2eq.framework.model.securityrules.SecurityContext.clear(); }` ----
- Realm routing: pass `X-Realm` in REST tests to select the target realm. `SecurityFilter` also validates realm access against user credentials when present.
- Data prep: If your test needs specific users/roles, create them under a `SecuritySession` beforehand (see `SecurityTest.ensureTestUserExists()`).
- Logging: enable `DEBUG` for `com.e2eq` to inspect rule evaluation and identity resolution during tests.

93.10. Summary

- Use `BaseRepoTest` for repository tests and migrations, and wrap work in `SecuritySession`.
- For less ceremony, create a simple scoped-call helper to run code under a `SecuritySession`.
- For REST/API tests and quick identity setup, use `@TestSecurity`, realm headers, and `RestAssured`.
- For full e2e security, obtain a JWT via the login API and include it in requests.

Chapter 94. 12. Next steps

- Explore the Supply Chain sample tutorial for an end-to-end scenario.
- Use the Reference Guide to quickly jump to deeper explanations of any topic above.