

Quantum Framework Documentation

Version 1.2.2-SNAPSHOT, 2025-11-13T18:20:58Z

Table of Contents

1. Getting Started	2
2. Overview: Building SaaS with Quantum	3
2.1. SaaS and Multi-Tenancy First	3
3. Getting Started: Your First Quantum Application	4
3.1. Prerequisites	4
3.2. Project Setup	4
3.3. Configuration	4
3.4. Your First Model	5
3.5. Repository	6
3.6. REST Resource	6
3.7. Running the Application	6
3.8. Testing Your API	7
3.9. What You Get Automatically	7
3.10. Next Steps	8
3.11. Optional: Enable Ontology Modules	8
4. Core Concepts	9
5. Multi-Tenancy Models	10
5.1. One Tenant per Database (in a MongoDB Cluster)	10
5.2. Many Tenants in One Database (Shared Database)	10
5.3. Freemium and Trial Tenants	10
6. DataDomain	12
7. DomainContext	13
8. RuleContext	14
9. End-to-End Flow	15
10. Resolvers and Variables in Rule Filters	16
11. Concrete example: building and using a resolver	17
11.1. 1) Implement the SPI	17
11.2. 2) How RuleContext uses resolvers	18
11.3. 3) Author a rule that consumes the variable	18
11.4. 4) End-to-end behavior	19
11.5. String literals vs. typed values in resolver variables	19
11.5.1. Example A: Resolver returns ObjectIds (typed)	19
11.5.2. Example B: Resolver returns String literals (force raw strings)	20
11.5.3. Other types supported	20
11.6. Using AccessListResolver with Ontology (optional)	21
11.6.1. Prefer Annotations for Functional Mapping (recommended)	22
11.6.2. DataDomain on Models	23
11.6.3. Persistence Repositories	24

11.6.4. Exposing REST Resources	24
11.6.5. Lombok in Models	25
11.6.6. Validation with Jakarta Bean Validation	25
11.6.7. Jackson vs Jakarta Validation Annotations	26
12. Jackson ObjectMapper in Quarkus and in Quantum	27
13. Validation Lifecycle and Morphia Interceptors	29
14. Functional Area/Domain in RuleContext Permission Language	31
15. StateGraphs on Models	34
16. State Graph (DOT)	35
17. CompletionTasks and CompletionTaskGroups	37
18. References and EntityReference	41
19. Tracking References with @TrackReferences and Delete Semantics	42
19.1. Ontologies in Quantum: Modeling Relationships That Are Resilient and Fast	43
19.1.1. What is an Ontology?	43
19.1.2. Ontology vs. Object Model	43
19.1.3. Why prefer Ontology-driven relationships over @Reference/EntityReference	44
19.1.4. How Quantum supports Ontologies	44
19.1.5. Modeling guidance: from object fields to predicates	45
19.1.6. Querying with ontology edges vs direct references	45
19.1.7. Migration: from @Reference to ontology edges	46
19.1.8. Performance and operational notes	46
19.1.9. How this integrates with Functional Areas/Domains	46
19.1.10. Summary	46
19.1.11. Concrete example: Sales Orders, Shipments, and evolving to Fulfillment>Returns ...	46
19.2. Integrating Ontology with Morphia, Permissions, and Multi-tenancy	51
19.2.1. Big picture: where ontology fits	51
19.2.2. Rule language: add hasEdge()	51
19.2.3. Passing tenantId correctly	52
19.2.4. Morphia repository integration patterns	52
19.2.5. Where to hook materialization	53
19.2.6. Security and multi-tenant considerations	53
19.2.7. Developer workflow and DX checklist	53
19.2.8. Cookbook: end-to-end example with Orders + Org	54
19.2.9. Migration notes for teams using @Reference	54
20. Building APIs	55
21. REST: Find, Get, List, Save, Update, Delete	56
21.1. Base Concepts	56
21.2. Example Resource	56
21.3. Authorization Layers in REST CRUD	56
21.4. Querying	57
21.5. Responses and Schemas	58

21.6. Error Handling	58
21.6.1. Query Language (ANTLR-based)	58
21.7. Simple filters (equals)	59
21.8. Advanced filters: grouping and AND/OR/NOT	59
21.9. IN lists	59
21.10. Sorting	60
21.11. Projections	60
21.12. End-to-end examples	60
22. CSV Export and Import	61
22.1. Export: GET /csv	61
22.2. Import: POST /csv (multipart)	63
22.3. Import with preview sessions	64
Authentication and Authorization	66
23. JWT Provider	67
24. Pluggable Authentication	68
25. Creating an Auth Plugin (using the Custom JWT provider as a reference)	69
26. AuthProvider interface (what a provider must implement)	70
27. UserManagement interface (operations your plugin must support)	71
28. Leveraging BaseAuthProvider in your plugin	72
29. Implementing your own provider	73
30. CredentialUserIdPassword model and DomainContext	74
31. Quarkus OIDC out-of-the-box and integrating with common IdPs	76
32. Authorization via RuleContext	78
32.1. Using Ontology Edges in List Endpoints (optional)	78
32.2. Basic Syntax	78
32.2.1. Operators	79
32.2.2. Value Types	79
32.2.3. Logical Operators	79
32.3. Common Patterns	80
32.3.1. String Matching	80
32.3.2. Numeric Ranges	80
32.3.3. Date and Time Queries	80
32.3.4. List Membership	80
32.3.5. Advanced Examples	81
32.3.6. Variables in Filters	82
32.3.7. Performance Tips	83
32.3.8. Integration with REST APIs	83
32.3.9. Error Handling	84
32.3.10. See Also	85
32.3.11. Execution engines and listeners	85
32.3.12. Expressing Ontology Constraints in Queries (optional)	88

32.4. Relationships and Ontology-aware Queries	89
32.4.1. Hydrating related data with expand(path)	89
32.4.2. Ontology operator: hasEdge(predicate, dst)	90
32.4.3. Root projection recap: fields:[+...,-...]	91
33. Relationship hydration with expand(path)	92
33.1. How it works	92
33.2. What is expand(path)?	92
33.3. Filtering related entities (inline)	92
33.4. Projection with fields:[+/-]	93
33.5. Examples	93
33.6. Semantics and limits	93
33.7. Aggregation examples: what expand(...) compiles to	94
33.7.1. Single reference: expand(customer)	94
33.7.2. Array of references: expand(items[*].product)	95
33.7.3. Nested arrays: expand(patient.visits[]. diagnoses [])	96
33.7.4. When is AGGREGATION used vs FILTER?	96
33.8. Planner decisions	97
33.9. Public facade: QueryGateway	97
33.10. Examples end-to-end	98
33.11. REST endpoints (v1)	98
33.11.1. Request/Response: /api/query/plan	98
33.11.2. Request/Response: /api/query/find	99
34. Security	101
Authentication and Authorization	102
35. JWT Provider	103
36. Pluggable Authentication	104
37. Creating an Auth Plugin (using the Custom JWT provider as a reference)	105
38. AuthProvider interface (what a provider must implement)	106
39. UserManagement interface (operations your plugin must support)	107
40. Leveraging BaseAuthProvider in your plugin	108
41. Implementing your own provider	109
42. CredentialUserIdPassword model and DomainContext	110
43. Quarkus OIDC out-of-the-box and integrating with common IdPs	113
44. Authorization via RuleContext	116
Permissions: Rule Bases, SecurityURIHeader, and SecurityURIBody	117
45. Introduction: Layered Enforcement Overview	118
46. Key Concepts	121
47. Rule Structure (Illustrative)	122
48. Matching Algorithm	124
49. Priorities	126
50. Grant-based vs Deny-based Rule Sets	127

51. Feature Flags, Variants, and Target Rules	129
52. Multiple Matching RuleBases	132
53. Identity and Role Matching	133
53.1. How roles are defined for an identity (role sources and resolution)	133
54. Example Scenarios	136
55. Operational Tips	137
56. How UIActions and DefaultUIActions are calculated	138
57. How This Integrates End-to-End	140
58. Administering Policies via REST (PolicyResource)	141
58.1. Model shape (Policy)	141
58.2. How rule-contributed filters work (andFilterString, orFilterString, joinOp)	141
58.3. AccessListResolvers (SPI) for list-based access	144
58.4. Endpoints	148
58.5. Examples	149
58.6. How changes affect rule bases and enforcement	150
59. Realm override (X-Realm) and Impersonation (X-Impersonate)	152
59.1. What they do (at a glance)	152
59.2. How the headers integrate with permission evaluation	152
59.3. Required credential configuration (CredentialUserIdPassword)	153
59.4. End-to-end behavior from SecurityFilter (reference)	154
59.5. Practical differences and use cases	154
59.6. Examples	154
60. Data domain assignment on create: DomainContext and DataDomainPolicy	156
60.1. The problem this solves (and why it matters)	156
60.2. Key concepts recap: DomainContext and DataDomain	156
60.3. The default policy (do nothing and it works)	156
60.4. Policy scopes: principal-attached vs. global	157
60.5. The policy map and matching	157
60.6. How the resolver works	158
60.7. When would you want a non-global policy?	159
60.8. Relation to tenancy models	159
60.9. Authoring tips	159
60.10. API pointers	160
60.11. Ontology in Permission Rules (optional)	160
60.11.1. Rule language: add hasEdge()	160
60.12. Label resolution SPI and hasLabel() in rules	163
61. Script Helpers reference (when enabled)	165
61.1. Security Annotations: FunctionalMapping and FunctionalAction	167
61.1.1. @FunctionalMapping	167
61.1.2. @FunctionalAction	167
61.1.3. Default Action Mapping	168

61.1.4. How the Framework Uses These Annotations	168
61.1.5. Migration from Legacy Methods	169
61.1.6. Best Practices	169
61.1.7. Integration with Permission Rules	170
61.1.8. See Also	170
62. Advanced Features	172
Database Migrations and Index Management	173
63. Overview	174
64. Semantic Versioning	175
65. Configuration	176
66. How change sets are discovered and executed	177
67. Authoring a change set	178
68. Example change sets in the framework	179
69. REST APIs to trigger migrations (MigrationResource)	180
70. Per-entity index management (BaseResource)	181
71. Global index management (MigrationService)	182
72. Validating versions at startup	183
73. Notes and best practices	184
73.1. Checksum vs. from/to versions	184
73.2. Testing Framework	185
73.3. Prerequisites and glossary	185
73.4. Pattern 1 — Extend BaseRepoTest	185
73.5. Pattern 2 — Try-with-resources SecuritySession	186
73.6. Pattern 3 — Scoped-call wrapper (ScopedCallScope)	187
73.7. Pattern 4 — @TestSecurity annotation (Quarkus)	188
73.8. Testing REST APIs vs. Repository Logic	189
73.9. Useful Quarkus Test features	189
73.10. Real-world tips	189
73.11. Summary	190
74. Tutorials	191
Supply Chain Collaboration SaaS: A Business-First Guide	192
75. What a supply-chain SaaS needs (and how Quantum helps)	193
76. Why multi-tenancy is a natural fit for supply chains	194
77. Who uses the system (organizations and roles)	195
78. Identity and access: meet partners where they are	196
79. Modeling without jargon: Areas, Domains, and Actions	197
80. Policies that say “who can do what” (Rule Language)	198
81. Domain contexts and domain policies (placing and sharing data)	199
82. Bootstrap tenants with Seed Packs (zero-touch baseline data)	200
83. Security policies: practical protections and proofs	201
84. A small, powerful API surface: List + Query	202

85. Delegated Administration (tenant-level user management)	203
86. Integrations and data management	204
87. End-to-end examples	205
88. What you don't have to build from scratch	206
89. Other helpful capabilities you get out of the box	207
90. Next steps	208
91. A day in the life: From Purchase Order to Delivery	209
92. Appendix	211
92.1. Configuration Reference	211
92.1.1. Configuration Basics	211
92.1.2. Database Configuration	211
92.1.3. Authentication Configuration	212
92.1.4. Quantum Framework Configuration	212
92.1.5. See Also	213
92.1.6. Ontology Configuration (optional)	213
92.1.7. Key Quarkus Capabilities	214
92.1.8. Arc (Quarkus CDI)	214
92.1.9. Native Compilation with GraalVM	215
92.1.10. Extension Ecosystem	216
92.1.11. Quantum's Quarkus Integration	216
92.1.12. Getting Started with Quarkus	216
92.1.13. See Also	216
92.1.14. Core Concepts	216
92.1.15. Multi-Tenancy	217
92.1.16. Security	217
92.1.17. Data and Persistence	218
92.1.18. REST and APIs	218
92.1.19. Framework Components	218
92.1.20. Annotations	219
92.1.21. External Integrations	219
92.1.22. Ontology (optional feature)	220

Quantum is a Quarkus-based framework for building multi-tenant SaaS platforms with MongoDB. This guide helps mid-level Java developers build secure, scalable applications with consistent REST APIs, policy-driven authorization, and flexible data sharing.

Chapter 1. Getting Started

Chapter 2. 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. 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. Getting Started: Your First Quantum Application

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

3.1. Prerequisites

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

3.2. 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>
```

3.3. 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
```

3.4. 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

3.5. 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
}

```

3.6. 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
}

```

3.7. 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/>

3.8. 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"
```

3.9. 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, createDate, 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

3.10. Next Steps

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

3.11. Optional: Enable Ontology Modules

You can adopt ontology incrementally. If you don't enable it, everything works as before.

Quick checklist

1) Add dependencies (app-level) - quantum-ontology-core, quantum-ontology-mongo, quantum-ontology-policy-bridge

2) Turn it on via config

```
e2eq.ontology.enabled=true
```

3) Provide an ontology registry (TBox) - Start with an in-memory registry (recommended initially). See [Ontologies in Quantum](#).

4) Wire data and indexes - Inject EdgeDao as a CDI bean (@Inject); indexes are ensured automatically at startup. - Indices: (tenantId, p, dst) and (tenantId, src, p)

5) Materialize edges on entity changes - Use OntologyMaterializer when sources or intermediates change (e.g., Order/Customer/Org/Address/Shipment in the e-commerce example).

6) Use edges in queries/policies - Wrap BSON via ListQueryRewriter or constrain by _id sets. See [Integrating Ontology](#).

Notes

- Keep it optional: only wire beans and create collections when e2eq.ontology.enabled=true.
- Multi-tenant: always pass tenantId from RuleContext.

Chapter 4. Core Concepts

Chapter 5. Multi-Tenancy Models

Quantum supports multiple multi-tenant models for MongoDB deployments:

5.1. 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.

5.2. 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.

5.3. 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. = DomainContext, RuleContext, and DataDomain

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

Chapter 6. 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 7. 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 8. 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 9. 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 10. 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 11. 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.

11.1. 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.

11.2. 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. Starting with this release you can reuse the exact same variable bundle for custom evaluation paths. The `RuleContext.resolveVariableBundle(...)` helper runs all matching `AccessListResolver` beans and returns the maps you can feed directly into `QueryPredicates` (for in-memory rules) or any other component that understands `${var}` tokens.

```
PrincipalContext pc = ...; // build from request
ResourceContext rc = ...;

MorphiaUtils.VariableBundle vars = ruleContext.resolveVariableBundle(pc, rc,
    UserProfile.class);

Predicate<JsonNode> predicate = QueryPredicates.compilePredicate(
    "customerId^[${accessibleCustomerIds}]",
    vars.strings,
    vars.objects
);

JsonNode order = QueryPredicates.toJsonNode(Map.of("customerId",
    "5f1e1a5e5e5e5e5e5e5e5e5e51"));
boolean allowed = predicate.test(order); // true when resolver returned that ObjectId
```

This allows the same resolver outputs to drive Morphia queries and client-side predicate checks without duplicating resolver logic.

11.3. 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`.

11.4. 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.

11.5. 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.

11.5.1. 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`.

11.5.2. 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(
                "5f1e1a5e5e5e5e5e5e5e5e51"),
            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).

11.5.3. 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.

11.6. Using AccessListResolver with Ontology (optional)

When ontology is enabled, an `AccessListResolver` can compute ID lists from semantic edges (materialized in Mongo) and publish them as variables for use in rule filters.

Example resolver (conceptual)

```
@ApplicationScoped
public class OrdersByOrgResolver implements AccessListResolver {
    @Inject EdgeDao edgeDao; // from quantum-ontology-mongo
    @Override public String key() { return "idsByPlacedInOrg"; }
    @Override public boolean supports(PrincipalContext p, ResourceContext r, Class<?
extends UnversionedBaseModel> model) {
        return model.getSimpleName().equals("Order");
    }
    @Override public Collection<?> resolve(PrincipalContext p, ResourceContext r, Class
<? extends UnversionedBaseModel> model) {
        String tenantId = p.getDataDomain().getTenantId();
        String org = p.getDataDomain().getOrgRefName();
        return edgeDao.srcIdsByDst(tenantId, "placedInOrg", org);
    }
}
```

Rule filter usage

```
id:^{idsByPlacedInOrg}
```

Notes

- Always scope by `tenantId` from `RuleContext/PrincipalContext`.
- This is optional and only active if you wire ontology components. For a deeper integration path, see [Integrating Ontology](#). == 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.

11.6.1. 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](#)

11.6.2. 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"; }
}

```

11.6.3. 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
}

```

11.6.4. 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.

11.6.5. 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.

11.6.6. 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.

11.6.7. 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 12. 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);
    }
}
```

```
}
```

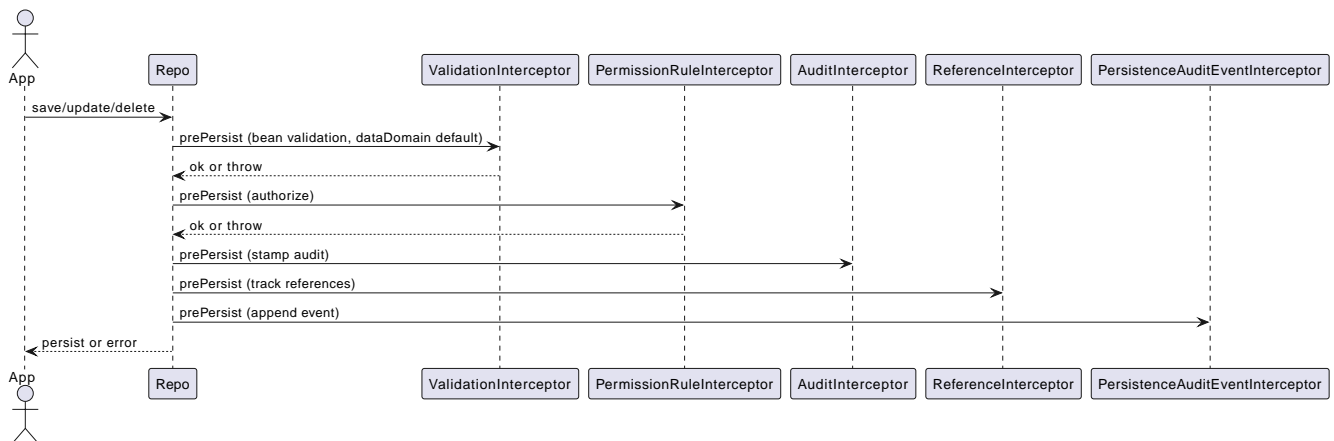
- Or set properties in `application.properties`:

```
# 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 13. Validation Lifecycle and Morphia Interceptors



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

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

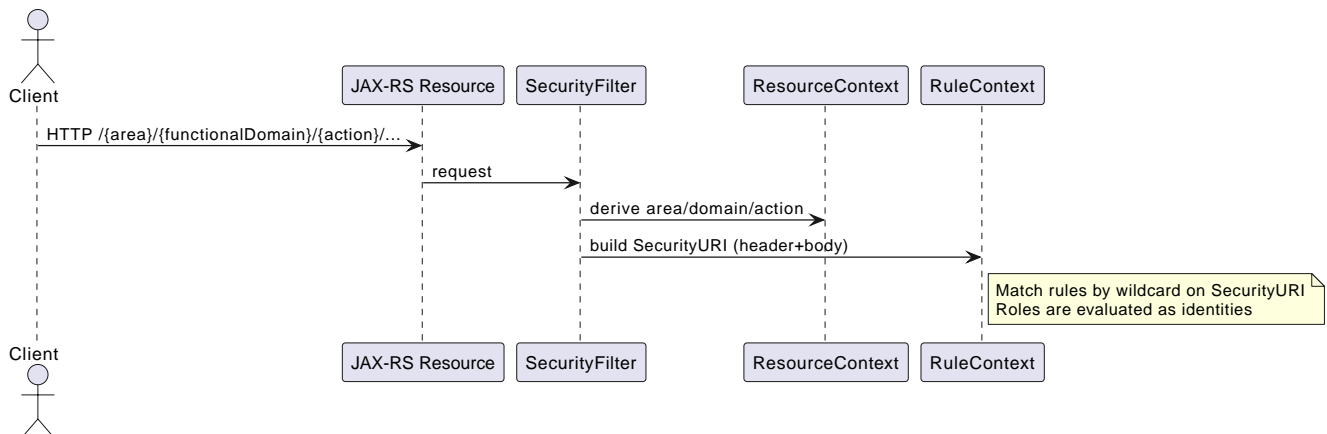
High-level responsibilities:

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

When does validation occur?

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

Chapter 14. 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 (SecurityURI-based matching)**: The framework derives area and functionalDomain from REST path segments using the convention: `/[area]/[functionalDomain]/[action]/...`. These values, together with action and identity, form the SecurityURI (header + body) used by the rule engine. Rules match on SecurityURI fields using wildcard string comparison; there is no HTTP method/URL pattern matching.
- **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) SecurityURI-based rule matching (Permissions)

```
- name: allow-catalog-product-reads
description: Allow USER and ADMIN to view products in the Catalog area
securityURI:
```

```

header:
  identity: USER          # role treated as identity; author a second rule for
ADMIN if needed
  area: Catalog
  functionalDomain: Product
  action: view
body:
  realm: system-com
  accountNumber: '*'
  tenantId: '*'
  dataSegment: '*'
  ownerId: '*'
  resourceId: '*'
effect: ALLOW
priority: 300
finalRule: false

```

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 15. 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:

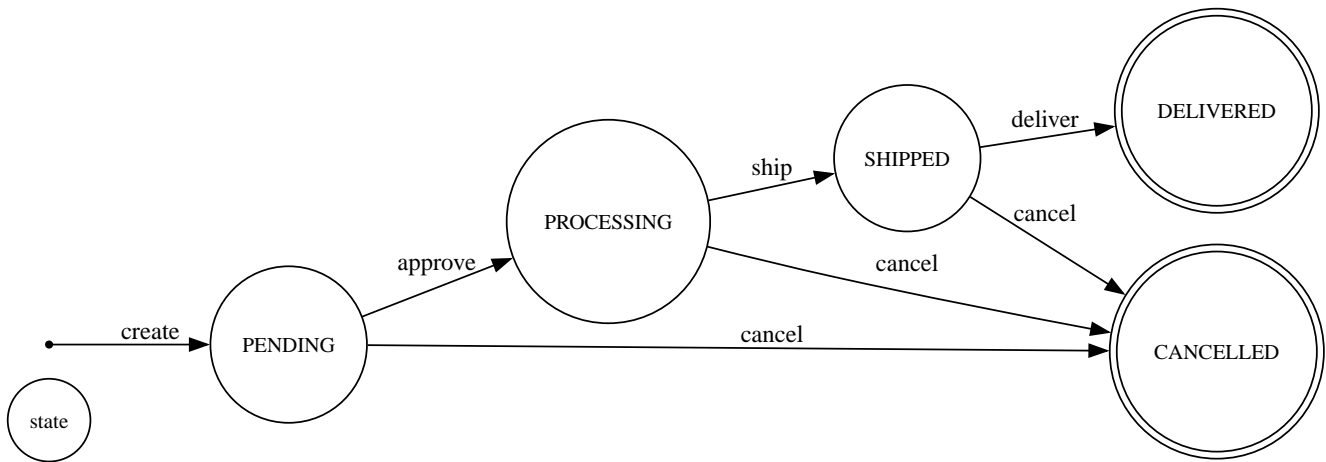
```
@Startup
@ApplicationScoped
public class StateGraphInitializer {
    @Inject StateGraphManager stateGraphManager;
    @PostConstruct void init() {
        StringState order = new StringState();
        order.setFieldName("orderStringState");

        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);
    }
}
```

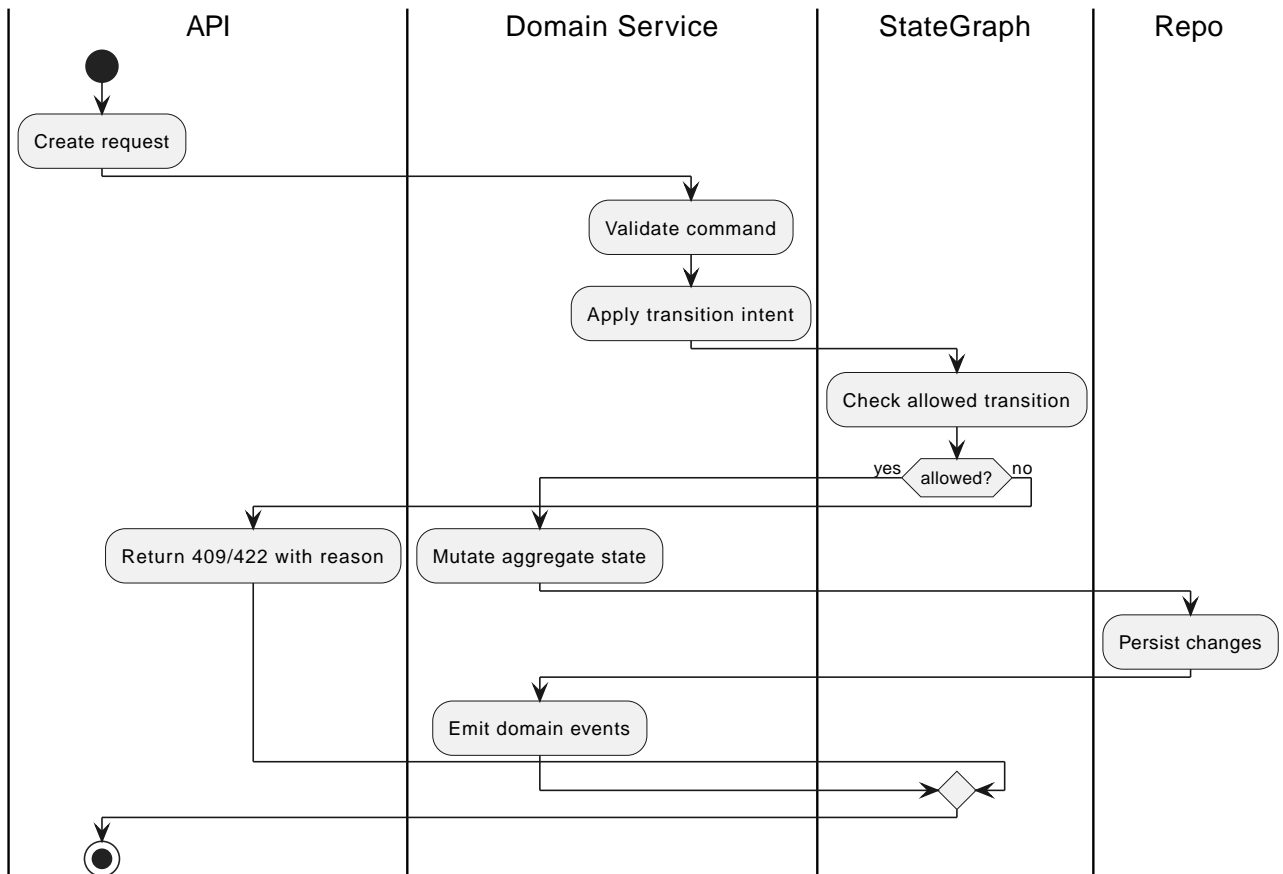
Chapter 16. State Graph (DOT)



Using the graph in a model:

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

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



How it affects save/update:

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

Chapter 17. 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

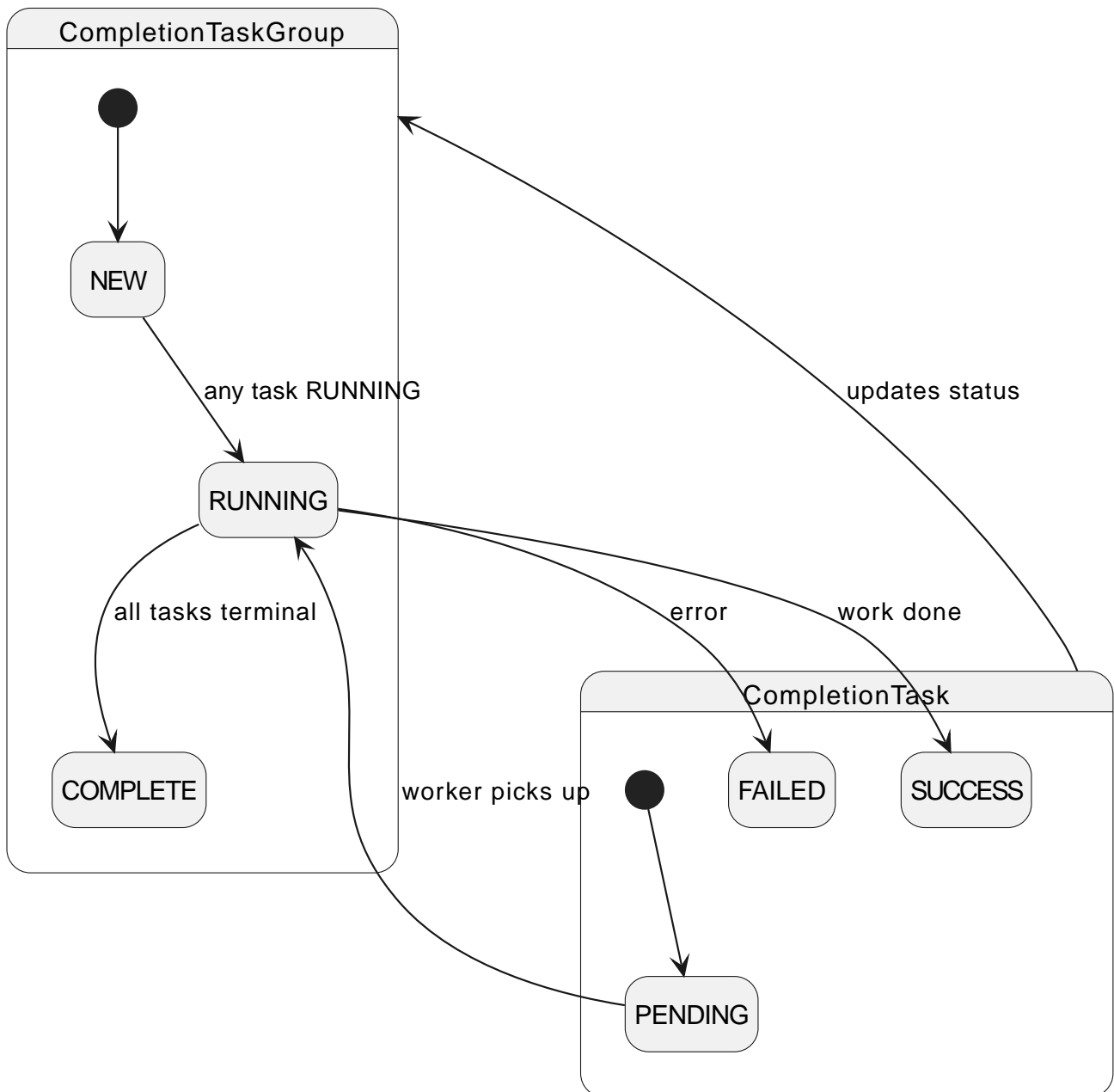
    @Override public String bmFunctionalArea() { return "TASK"; }
    @Override public String bmFunctionalDomain() { return "COMPLETION_TASK"; }
}

// 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 18. 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 19. 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.

19.1. Ontologies in Quantum: Modeling Relationships That Are Resilient and Fast



Looking for the short implementation plan? See PROPOSAL.md at the repository root for a concise module-by-module checklist.

This section explains what an ontology is, how it differs from a traditional object model, and how the Quantum Ontology modules make it practical to apply ontology ideas to your domain models and queries. It also contrasts ontology-driven relationships with direct object references (for example, using `@Reference` or `EntityReference`).

19.1.1. What is an Ontology?

In software terms, an ontology is a formal, explicit specification of concepts and their relationships.

- Concepts (Classes): Named categories/types in your domain. Concepts can form taxonomies (is-a hierarchies), be declared disjoint, or be equivalent.
- Relationships (Properties): Named relationships between entities. Properties can have a domain (applies to X) and a range (points to Y). They may be inverse or transitive.
- Axioms (Rules): Constraints and entailment rules, including property chains such as: if $(A \xrightarrow{p} B)$ and $(B \xrightarrow{q} C)$ then we infer $(A \xrightarrow{r} C)$.
- Inference: The process of deriving new facts (types, labels, edges) that were not explicitly stored but follow from axioms and known facts.

An ontology is not the data; it is the schema plus logic that gives your data additional meaning and enables consistent, automated inferences.

19.1.2. Ontology vs. Object Model

A conventional object model focuses on concrete classes, fields, and direct references between objects at implementation time. An ontology focuses on semantic types and relationships, with explicit rules that can derive new knowledge independent of how objects are instantiated.

Key differences:

- Purpose - Object model: Encapsulate data and behavior for application code generation and persistence.
- Ontology: Encode shared meaning, constraints, and inference rules that remain stable as implementation details change.
- Relationship handling - Object model: Typically uses direct references or foreign keys; traversals are hard-coded and fragile to change.
- Ontology: Uses named predicates (properties) and can infer additional relationships by rules (property chains, inverses, transitivity).
- Polymorphism and evolution - Object model: Polymorphism requires class inheritance in code; cross-cutting categories are awkward to add later.
- Ontology: Entities can have multiple types/labels at once. New concepts and properties can be introduced without breaking existing data.
- Querying - Object model: Queries couple to concrete classes and field paths; changes force query rewrites.
- Ontology: Queries target semantic

relationships; reasoners can materialize edges that queries reuse, decoupling queries from implementation details.

19.1.3. Why prefer Ontology-driven relationships over @Reference/EntityReference

Direct references (@Reference or custom EntityReference) are simple to start but become restrictive as domains grow: - Tight coupling: Code and queries couple to concrete field paths (customer.primaryAddress.id), making refactors risky. - Limited expressivity: Hard to encode and reuse higher-order relationships (e.g., "partners of my supplier's parent org"). - Poor polymorphism: References point to one collection/type; accommodating multiple target types requires extra code. - Performance pitfalls: Deep traversals cause extra queries, N+1 selects, or complex \$lookup joins.

Ontology-driven edges address these issues: - Decoupling via predicates: Use named predicates (e.g., hasAddress, memberOf, supplies) that remain stable while internal object fields change. - Inference for reachability: Property chains can materialize implied links ($A \xrightarrow{p} B \ \& \ B \xrightarrow{q} C \Rightarrow A \xrightarrow{r} C$), avoiding runtime multi-hop traversals. - Polymorphism-first: A predicate can connect heterogeneous types; type inferences (domain/range) remain consistent. - Query performance: Pre-materialized edges allow single-hop, index-friendly queries (in or eq filters) instead of ad-hoc multi-collection traversals. - Resilience to change: You can add or modify rules without rewriting data structures or touching referencing fields across models.

19.1.4. How Quantum supports Ontologies

Quantum provides three cooperating modules that make ontology modeling practical and fast:

- quantum-ontology-core (package com.e2eq.ontology.core)
- OntologyRegistry: Holds the TBox (terminology) of your ontology.
- ClassDef: Concept names and relationships (parents, disjointWith, sameAs).
- PropertyDef: Property names with optional domain, range, inverse flags, and transitivity.
- PropertyChainDef: Rules that define multi-hop implications (chains \rightarrow implied property).
- TBox: Container for classes, properties, and property chains.
- Reasoner interface and ForwardChainingReasoner: Given an entity snapshot and the registry, computes inferences:
- New types/labels to assert on entities.
- New edges to add (implied by property chains, inverses, or other rules).
- quantum-ontology-mongo (package com.e2eq.ontology.mongo)
- EdgeDao: A thin DAO around an edges collection in Mongo. Each edge contains tenantId, src, predicate p, dst, inferred flag, provenance, and timestamp.
- OntologyMaterializer: Runs the Reasoner for an entity snapshot and upserts the inferred edges, so queries can be rewritten to simple in/in eq filters.
- quantum-ontology-policy-bridge (package com.e2eq.ontology.policy)
- ListQueryRewriter: Takes a base query and rewrites it using the EdgeDao to filter by the set of

source entity ids that have a specific predicate to a given destination.

- This integrates ontology edges with RuleContext or policy decisions: policy asks for entities related by a predicate; the rewriter converts that into an efficient Mongo query.

These modules let you define your ontology (core), materialize derived relations (mongo), and leverage them in access and list queries (policy bridge).

19.1.5. Modeling guidance: from object fields to predicates

- Name relationships explicitly
- Define clear predicate names (hasAddress, memberOf, supplies, owns, assignedTo). Avoid encoding relationship semantics in field names only.
- Keep object model minimal and flexible
- Store lightweight identifiers (ids) as needed, but avoid deeply nested reference graphs that encode traversals in code.
- Model polymorphic relationships
- Prefer predicates that naturally connect multiple possible types (e.g., assignedTo can target User, Team, Bot) and rely on ontology type assertions to constrain where needed.
- Use property chains for common paths
- If business logic often traverses $A \rightarrow B \rightarrow C$, define a chain $p \sqcap q \Rightarrow r$ and materialize r for faster queries and simpler policies.
- Capture inverses and transitivity
- For natural inverses (parentOf \sqcap childOf) or transitive relations (partOf, locatedIn), define them in the ontology so edges and queries stay consistent.
- Keep provenance
- Record why an edge exists (prov.rule, prov.inputs) so you can recompute, audit, or retract when inputs change.

19.1.6. Querying with ontology edges vs direct references

- Direct reference example (fragile/slow)
- Query: "Find Orders whose buyer belongs to Org X or its parents."
- With @Reference: requires joining $\text{Order} \rightarrow \text{User} \rightarrow \text{Org}$ and recursing org.parent; costly and tightly coupled to fields.
- Ontology edge example (resilient/fast)
- Define predicates: placedBy(order, user), memberOf(user, org), ancestorOf(org, org). Define chain placedBy \sqcap memberOf \Rightarrow placedInOrg.
- Materialize edges: (order --placedInOrg--> org). Also make ancestorOf transitive.
- Query becomes: where order._id in EdgeDao.srcIdsByDst(tenantId, "placedInOrg", orgX).
- With transitivity, you can precompute ancestor closure or add a chain placedInOrg \sqcap ancestorOf \Rightarrow placedInOrg to include parents automatically.

19.1.7. Migration: from @Reference to ontology edges

- Start by introducing predicates alongside existing references; do not remove references immediately.
- Materialize edges for hot read paths; keep provenance so you can reconstruct.
- Gradually update queries (list screens, policy filters) to use ListQueryRewriter with EdgeDao instead of deep traversals or \$lookup.
- Once stable, you can simplify models by removing rigid reference fields where unnecessary and rely on edges for read-side composition.

19.1.8. Performance and operational notes

- Indexing: Create compound indexes on edges: (tenantId, p, dst) and (tenantId, src, p) to support both reverse and forward lookups.
- Write amplification vs read wins: Materialization adds write work, but dramatically improves read latency and simplifies queries.
- Consistency: Re-materialize edges on relevant entity changes (source, destination, or intermediate) using OntologyMaterializer.
- Multi-tenancy: Keep tenantId in the edge key and filters; the provided EdgeDao methods include tenant scoping.

19.1.9. How this integrates with Functional Areas/Domains

- Functional domains often map to concept clusters in the ontology. Use @FunctionalMapping to aid discovery and apply policies per area/domain.
- Policies can refer to relationships semantically ("hasEdge placedInOrg OrgX") and rely on the policy bridge to turn this into efficient data filters.

19.1.10. Summary

- Ontology-powered relationships provide a stable, semantic layer over your object model.
- The Quantum Ontology modules let you define, infer, and query these relationships efficiently on MongoDB.
- Compared with direct @Reference/EntityReference, ontology edges are more expressive, resilient to change, and typically faster for complex list/policy queries once materialized.

19.1.11. Concrete example: Sales Orders, Shipments, and evolving to Fulfillment>Returns

This example shows how to use an ontology to model relationships around Orders, Customers, and Shipments, and how the model can evolve to include Fulfillment and Returns without breaking existing queries. We will:

- Define core concepts and predicates.
- Add property chains that materialize implied relationships for fast queries.

- Show how queries are rewritten using edges instead of deep object traversals.
- Evolve the model to support Fulfillment and Returns with minimal changes.

Core concepts (classes)

- Order, Customer, Organization, Shipment, Address, Region
- Later evolution: FulfillmentTask, FulfillmentUnit, ReturnRequest, ReturnItem, RMA

Key predicates (relationships)

- placedBy(order, customer): who placed the order
- memberOf(customer, org): a customer belongs to an organization (or account)
- orderHasShipment(order, shipment): outbound shipment for the order
- shipsTo(shipment, address): shipment destination
- locatedIn(address, region): address is located in a Region
- ancestorOf(org, org): organizational ancestry (transitive)

Property chains (implied relationships)

- placedBy \square memberOf \Rightarrow placedInOrg
- If (order --placedBy- \rightarrow customer) and (customer --memberOf- \rightarrow org), then infer (order --placedInOrg- \rightarrow org)
- orderHasShipment \square shipsTo \Rightarrow orderShipsTo
- If (order --orderHasShipment- \rightarrow shipment) and (shipment --shipsTo- \rightarrow address), infer (order --orderShipsTo- \rightarrow address)
- orderShipsTo \square locatedIn \Rightarrow orderShipsToRegion
- If (order --orderShipsTo- \rightarrow address) and (address --locatedIn- \rightarrow region), infer (order --orderShipsToRegion- \rightarrow region)
- placedInOrg \square ancestorOf \Rightarrow placedInOrg
- Makes placedInOrg resilient to org hierarchy changes (ancestorOf is transitive). This is a common “closure” trick: re-assert the same predicate via chain to absorb hierarchy.

A minimal Java-style snippet to define this TBox

```
import java.util.*;
import com.e2eq.ontology.core.OntologyRegistry;
import com.e2eq.ontology.core.OntologyRegistry.*;

Map<String, ClassDef> classes = Map.of(
    "Order", new ClassDef("Order", Set.of(), Set.of(), Set.of()),
    "Customer", new ClassDef("Customer", Set.of(), Set.of(), Set.of()),
    "Organization", new ClassDef("Organization", Set.of(), Set.of(), Set.of()),
    "Shipment", new ClassDef("Shipment", Set.of(), Set.of(), Set.of()),
    "Address", new ClassDef("Address", Set.of(), Set.of(), Set.of()),
```

```

    "Region", new ClassDef("Region", Set.of(), Set.of(), Set.of())
);

Map<String, PropertyDef> props = Map.of(
    "placedBy", new PropertyDef("placedBy", Optional.of("Order"), Optional.of("Customer"), false, Optional.empty(), false),
    "memberOf", new PropertyDef("memberOf", Optional.of("Customer"), Optional.of("Organization"), false, Optional.empty(), false),
    "orderHasShipment", new PropertyDef("orderHasShipment", Optional.of("Order"), Optional.of("Shipment"), false, Optional.empty(), false),
    "shipsTo", new PropertyDef("shipsTo", Optional.of("Shipment"), Optional.of("Address"), false, Optional.empty(), false),
    "locatedIn", new PropertyDef("locatedIn", Optional.of("Address"), Optional.of("Region"), false, Optional.empty(), false),
    "ancestorOf", new PropertyDef("ancestorOf", Optional.of("Organization"), Optional.of("Organization"), false, Optional.empty(), true), // transitive
    // implied predicates (no domain/range required, but you may add them for validation)
    "placedInOrg", new PropertyDef("placedInOrg", Optional.of("Order"), Optional.of("Organization"), false, Optional.empty(), false),
    "orderShipsTo", new PropertyDef("orderShipsTo", Optional.of("Order"), Optional.of("Address"), false, Optional.empty(), false),
    "orderShipsToRegion", new PropertyDef("orderShipsToRegion", Optional.of("Order"), Optional.of("Region"), false, Optional.empty(), false)
);

List<PropertyChainDef> chains = List.of(
    new PropertyChainDef(List.of("placedBy", "memberOf"), "placedInOrg"),
    new PropertyChainDef(List.of("orderHasShipment", "shipsTo"), "orderShipsTo"),
    new PropertyChainDef(List.of("orderShipsTo", "locatedIn"), "orderShipsToRegion"),
    new PropertyChainDef(List.of("placedInOrg", "ancestorOf"), "placedInOrg")
);

OntologyRegistry.TBox tbox = new OntologyRegistry.TBox(classes, props, chains);
OntologyRegistry registry = OntologyRegistry.inMemory(tbox);

```

Materializing edges for an Order

- Explicit facts for order O1:
- O1 placedBy C9
- C9 memberOf OrgA
- O1 orderHasShipment S17
- S17 shipsTo Addr42
- Addr42 locatedIn RegionWest
- OrgA ancestorOf OrgParent
- Inferred edges after running the reasoner for O1's snapshot:
- O1 placedInOrg OrgA

- O1 placedInOrg OrgParent (via closure with ancestorOf)
- O1 orderShipsTo Addr42
- O1 orderShipsToRegion RegionWest

How queries become simple and fast

- List Orders for Organization OrgParent (including children):
- Instead of joining Order → Customer → Org and recursing org.parent, run a single filter using materialized edges.

```
import com.mongodb.client.model.Filters;
import org.bson.conversions.Bson;
import com.e2eq.ontology.policy.ListQueryRewriter;

Bson base = Filters.eq("status", "OPEN");
Bson rewritten = rewriter.rewriteForHasEdge(base, tenantId, "placedInOrg", "OrgParent");
// Use rewritten in your Mongo find
```

- List Orders shipping to RegionWest:

```
Bson rewritten2 = rewriter.rewriteForHasEdge(Filters.empty(), tenantId,
"orderShipsToRegion", "RegionWest");
```

Why this is resilient

- If tomorrow Customer becomes AccountContact and the organization model gains Divisions and multi-parent org graphs, you only adjust predicates and chains.
- Queries that rely on placedInOrg or orderShipsToRegion remain unchanged and fast, because edges are re-materialized by OntologyMaterializer.

Evolving the model: add Fulfillment

New concepts

- FulfillmentTask: a unit of work to pick/pack/ship order lines
- FulfillmentUnit: a logical grouping (e.g., wave, tote, parcel)

New predicates

- fulfills(task, order)
- realizedBy(order, fulfillmentUnit)
- taskProduces(task, shipment)

New chains (implied)

- fulfills \Rightarrow derived edge from task to order; combine with taskProduces to connect order to shipment without touching Order fields:
- fulfills \square taskProduces \Rightarrow orderHasShipment
- realizedBy \square orderHasShipment \Rightarrow fulfilledByUnit
- If (order --realizedBy- \rightarrow fu) and (order --orderHasShipment- \rightarrow s) \Rightarrow (fu --fulfillsShipment- \rightarrow s) or simply (order --fulfilledByUnit- \rightarrow fu)

These chains let you introduce warehouse concepts without changing how UI filters orders by organization or ship-to region. Existing queries still operate via placedInOrg and orderShipsToRegion.

Evolving further: add Returns

New concepts

- ReturnRequest, ReturnItem, RMA

New predicates

- hasReturn(order, returnRequest)
- returnFor(returnItem, order)
- returnRma(returnRequest, rma)

New chains (implied)

- hasReturn \Rightarrow openReturnOnOrg via placedInOrg:
- hasReturn \square placedInOrg \Rightarrow returnPlacedInOrg
- returnFor \square orderShipsToRegion \Rightarrow returnShipsToRegion

Example queries with new capabilities

- List Orders with open returns in OrgParent:

```
Bson r = rewriter.rewriteForHasEdge(Filters.empty(), tenantId, "returnPlacedInOrg",
"OrgParent");
```

- List Returns associated to Orders shipping to RegionWest:

```
Bson r2 = rewriter.rewriteForHasEdge(Filters.empty(), tenantId, "returnShipsToRegion",
"RegionWest");
```

Comparison with direct references (@Reference/EntityReference)

- With direct references you would encode fields like Order.customer, Order.shipments, Shipment.address, Address.region and then implement multi-hop traversals in code or \$lookup pipelines, rewriting them whenever you add Fulfillment or Returns.

- With ontology edges, you keep predicates stable and add property chains. Existing list and policy queries keep working and typically become faster due to single-hop filters on an indexed edges collection.

Operational tips for this scenario

- Ensure EdgeDao has indexes on (tenantId, p, dst) and (tenantId, src, p).
- Use OntologyMaterializer when Order, Shipment, Customer, Address, or org hierarchy changes to keep edges fresh.
- Keep provenance in edge.prov (rule, inputs) so you can recompute or retract edges when source data changes.

19.2. Integrating Ontology with Morphia, Permissions, and Multi-tenancy

This section focuses on integration and developer experience: how ontology edges flow into Morphia-based repositories and the permission rule language, while remaining fully multi-tenant and secure.

19.2.1. Big picture: where ontology fits

- Write path (materialization):
- Your domain code persists entities with minimal direct references.
- An OntologyMaterializer runs when entities change to derive and upsert edges into the edges collection (per tenant).
- Policy path (authorization and list filters):
- The permission rule language evaluates the caller's SecurityContext/RuleContext and produces logical filters.
- When a rule asks for a semantic relationship (hasEdge), we use ListQueryRewriter + EdgeDao to translate that into efficient Mongo filters over ids.
- Read path (queries):
- Morphia repos apply the base data-domain filters and the rewritten ontology constraint to queries, producing fast lists without deep joins.

19.2.2. Rule language: add hasEdge()

We introduce a policy function/operator to reference ontology edges directly from rules:

- Signature: hasEdge(predicate, dstIdOrVar)
- predicate: String name of the ontology predicate (e.g., "placedInOrg", "orderShipsToRegion").
- dstIdOrVar: Either a concrete id/refName or a variable resolved from RuleContext (e.g., principal.orgRefName, request.region).
- Semantics: The rule grants/filters entities for which an edge (tenantId, src = entity_id, p =

predicate, dst = resolvedDst) exists.

- Composition: hasEdge can be combined with existing rule clauses (and/or/not) and other filters (states, tags, ownerId, etc.).

Example rule snippets (illustrative):

- Allow viewing Orders in the caller's org (including ancestors via ontology closure):
- allow VIEW Order when hasEdge("placedInOrg", principal.orgRefName)
- Restrict list to Orders shipping to a region chosen in request:
- allow LIST Order when hasEdge("orderShipsToRegion", request.region)

Under the hood, policy evaluation uses `ListQueryRewriter.rewriteForHasEdge(...)`, which converts hasEdge into a set of source ids and merges that with the base query.

19.2.3. Passing tenantId correctly

- Always resolve tenantId from RuleContext/SecurityContext (the same source your repos use for realm/database selection).
- EdgeDao and ListQueryRewriter already accept tenantId; never cross tenant boundaries when reading edges.
- Index recommendation (per tenant):
- (tenantId, p, dst)
- (tenantId, src, p)

19.2.4. Morphia repository integration patterns

The goal is zero-friction usage in existing repos without invasive changes.

Option A: Apply ontology constraints in code paths that already construct BSON filters.

- If your repo method builds a Bson filter before calling find(), wrap it through rewriter:

```
Bson base = Filters.and(existingFilters...);
Bson rewritten = hasEdgeRequested
    ? rewriter.rewriteForHasEdge(base, tenantId, predicate, dst)
    : base;
var cursor = datastore.getDatabase().getCollection(coll).find(rewritten);
```

Option B: Apply ontology constraints to Morphia Filter/Query via ids.

- When the repo uses Morphia's typed query API instead of BSON, pre-compute the id set and constrain by `_id`:

```
Set<String> ids = edgeDao.srcIdsByDst(tenantId, predicate, dst);
if (ids.isEmpty()) {
```

```

return List.of(); // short-circuit
}
query.filter(Filters.in("_id", ids));

```

Option C: Centralize in a tiny helper for developer ergonomics.

- Provide one helper in your application layer, invoked wherever policies inject additional constraints:

```

public final class OntologyFilterHelper {
    private final ListQueryRewriter rewriter;
    public OntologyFilterHelper(ListQueryRewriter r) { this.rewriter = r; }

    public Bson ensureHasEdge(Bson base, String tenantId, String predicate, String dst)
    {
        return rewriter.rewriteForHasEdge(base, tenantId, predicate, dst);
    }
}

```

19.2.5. Where to hook materialization

- On entity changes that are sources or intermediates for chains:
- Order (placedBy, orderHasShipment), Customer (memberOf), Shipment (shipsTo), Address (locatedIn), Organization (ancestorOf/parent), and any Fulfillment/Returns entities.
- Recommended patterns:
- On-save/on-update hooks in your service layer call `OntologyMaterializer.apply(...)` with the explicit edges known from the entity snapshot.
- For intermediates (e.g., `Address.region` changed), enqueue affected sources for recomputation; use provenance to locate impacted edges.
- Provide nightly/backfill jobs for recomputing edges across a tenant when ontology rules evolve.

19.2.6. Security and multi-tenant considerations

- Edge rows include `tenantId` and should be validated/filtered by tenant on every operation.
- Never trust a client-supplied predicate or destination id blindly; combine with rule evaluation and whitelist allowed predicates per domain if needed.
- For shared resources across tenants (rare), model cross-tenant permissions at the policy layer; don't reuse edges across tenants unless explicitly designed.

19.2.7. Developer workflow and DX checklist

- When writing a rule: use `hasEdge("<predicate>", <rhs>)` and rely on `RuleContext` variables for the destination when possible.
- When writing a list endpoint: read optional ontology filter hints from the policy layer; if

present, apply `ensureHasEdge(...)` before `find()`.

- When changing domain relationships: update predicates/chains and re-materialize; list/policy code stays unchanged.
- When indexing a new tenant: include the edges indexes early and validate via a smoke test query using `ListQueryRewriter`.

19.2.8. Cookbook: end-to-end example with Orders + Org

- Policy: allow LIST Order when `hasEdge("placedInOrg", principal.orgRefName)`
- Request lifecycle: 1) Security filter builds `SecurityContext` and `RuleContext` with `tenantId` and `principal`. 2) Policy evaluation returns a directive to constrain by `hasEdge("placedInOrg", orgRefName)`. 3) Repo builds base filter (`state != ARCHIVED`, etc.). 4) Repo calls `OntologyFilterHelper.ensureHasEdge(base, tenantId, "placedInOrg", orgRefName)`. 5) Mongo executes a single-hop query using materialized edges; results respect both policy and multi-tenancy.

19.2.9. Migration notes for teams using @Reference

- Keep existing references for write-side integrity and local joins where simple.
- Introduce ontology edges on hot read paths first; update policy rules to `hasEdge` and verify results.
- Gradually replace deep `$lookup` traversals with `hasEdge`-based rewrites.
- Ensure materialization hooks are deployed before removing data fields used as inputs to the ontology.

Chapter 20. Building APIs

Chapter 21. REST: Find, Get, List, Save, Update, Delete

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

21.1. 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.

21.2. Example Resource

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

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

21.3. 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.

21.4. 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.

21.5. 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.

21.6. 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.

21.6.1. Query Language (ANTLR-based)

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

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

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

21.7. Simple filters (equals)

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

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

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

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

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

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

21.9. IN lists

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

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

21.10. Sorting

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

Examples:

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

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

21.11. Projections

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

Examples:

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

21.12. End-to-end examples

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

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

Chapter 22. 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.

22.1. 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"
```

22.2. 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"
```

22.3. 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 23. 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 24. 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 25. 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 26. 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 27. 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 28. 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 29. 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 30. 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 31. 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 32. 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.

32.1. Using Ontology Edges in List Endpoints (optional)

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

Pattern A: Wrap BSON with ListQueryRewriter

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

Pattern B: Constrain Morphia query by IDs

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

Notes

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

32.2. Basic Syntax

32.2.1. 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"]

32.2.2. 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
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}

32.2.3. 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

32.3. Common Patterns

32.3.1. 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"
```

32.3.2. Numeric Ranges

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

# Quantity greater than 0
quantity:>#0

# Exact count
itemCount:#5
```

32.3.3. 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
```

32.3.4. 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:!""

```

32.3.5. 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
```

32.3.6. 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}]
```

32.3.7. 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
```

32.3.8. 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=-createdAt&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})"
```

32.3.9. 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)
```

32.3.10. See Also

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

32.3.11. Execution engines and listeners

The BIAPI query syntax is parsed once (via ANTLR) and can be executed by different "listeners" depending on the use case. Quantum ships with two primary implementations that share the same grammar and semantics:

- Morphia listener: converts a query into Mongo/Morphia Filters for database-side execution
- In-memory listener: converts a query into a Java Predicate over JSON data for Quarkus/GraalVM-friendly in-memory execution

Morphia: QueryToFilterListener

Use this when you want the database to perform the filtering. The listener walks the parse tree and produces a `dev.morphia.query.filters.Filter` which you can apply to Morphia queries. This is ideal for repository APIs and any endpoint where you want to leverage MongoDB indexes and avoid loading large data sets into memory.

Key characteristics: - Output type: Morphia Filter - Execution: database-side (MongoDB) - Semantics: identical to grammar (comparisons, IN/NIN, exists, null, regex with wildcards, `elemMatch`, boolean `&&/||/!!`) - Variable expansion: supports `${vars}` and single-variable IN list expansion (e.g., `[$ids]` can expand to a collection/array or a comma-separated string)

Example:

```
import com.e2eq.framework.grammar.*;
import com.e2eq.framework.model.persistent.morphia.QueryToFilterListener;
import dev.morphia.query.filters.Filter;
import dev.morphia.query.filters.Filters;
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.ParseTreeWalker;
import org.apache.commons.text.StringSubstitutor;

String query = "(status:Assigned||status:Pending)&&displayName:*Route*";
var vars = java.util.Map.<String,String>of();

// Parse
CharStream cs = CharStreams.fromString(query);
BIAPIQueryLexer lexer = new BIAPIQueryLexer(cs);
CommonTokenStream tokens = new CommonTokenStream(lexer);
BIAPIQueryParser parser = new BIAPIQueryParser(tokens);
```

```

BIAPIQueryParser.QueryContext tree = parser.query();

// Build Morphia filter
QueryToFilterListener listener = new QueryToFilterListener(vars, new
StringSubstitutor(vars), /* modelClass */ null);
ParseTreeWalker.DEFAULT.walk(listener, tree);
Filter morphiaFilter = listener.getFilter();

// Use with Morphia query (example)
// datastore.find(MyEntity.class).filter(morphiaFilter).iterator().toList();

```

A few query examples (taken from testQueryStrings.txt):

- Equality: field:"quotedString"
- Comparisons: field:>##12.56, field:<=#123
- IN/NIN: field:^[value1,value2], field:!^[value1,value2]
- Exists/Null: field:~, field:null
- elemMatch: arrayField: {(subField:<#12) | (subField:>#15)}

In-memory (JsonNode): QueryToPredicateJsonListener

Use this when you need to evaluate queries in memory without reflection on POJOs. This implementation compiles a query into a `java.util.function.Predicate` over a Jackson `JsonNode`. It is Quarkus/GraalVM friendly, useful for: - Unit tests where you want to validate query behavior without a database - Post-filtering or pre-filtering of already-fetched data - Evaluating access rules or business logic against transient objects

Key characteristics: - Output type: `Predicate<JsonNode>` - Execution: in-memory - No runtime reflection: operates on `JsonNode` - Semantics and variable expansion match the Morphia listener

Convenience helpers exist in `QueryPredicates`:

```

import com.e2eq.framework.query.QueryPredicates;
import com.fasterxml.jackson.databind.JsonNode;
import java.util.function.Predicate;
import java.util.Map;

String query = "(status:Assigned||status:Pending)&&displayName:*Route*";
Predicate<JsonNode> p = QueryPredicates.compilePredicate(query, Map.of(), Map.of());

// Example data as a POJO or Map -> convert to JsonNode
record Ticket(String status, String displayName) {}
Ticket ticket = new Ticket("Assigned", "Route Exception in
Route:To[http://com.xxx/update]");
JsonNode node = QueryPredicates.toJsonNode(ticket);

boolean include = p.test(node); // true

```

Additional examples

- Equality and comparisons

```
var vars = Map.<String,String>of();
var objVars = Map.<String,Object>of();
Predicate<JsonNode> eq = QueryPredicates.compilePredicate("quantity:#42", vars,
objVars);
Predicate<JsonNode> gt = QueryPredicates.compilePredicate("price:>##19.99", vars,
objVars);

JsonNode product = QueryPredicates.toJsonNode(Map.of("quantity", 42, "price", 25.00));
assert eq.test(product);
assert gt.test(product);
```

- IN / NIN with variable expansion

```
var vars = Map.of("principalId", "66d1f1ab452b94674bbd934a");
Predicate<JsonNode> in =
QueryPredicates.compilePredicate("ownerId:^([${principalId}],value2]", vars, Map.of());
JsonNode doc = QueryPredicates.toJsonNode(Map.of("ownerId",
"66d1f1ab452b94674bbd934a"));
assert in.test(doc);
```

- elemMatch over arrays of objects

```
String q = "items:{{(sku:abc||qty:>#10)&&price:<=##9.99}}";
Predicate<JsonNode> em = QueryPredicates.compilePredicate(q, Map.of(), Map.of());
JsonNode order = QueryPredicates.toJsonNode(Map.of(
    "items", java.util.List.of(
        Map.of("sku","abc","qty", 5, "price", 9.99),
        Map.of("sku","xyz","qty", 12, "price", 8.50)
    )));
// Matches: first item by sku OR second item by qty with price cap
assert em.test(order);
```

- Regex with wildcards

```
Predicate<JsonNode> rx = QueryPredicates.compilePredicate("displayName:*Route*",
Map.of(), Map.of());
JsonNode ticket = QueryPredicates.toJsonNode(Map.of("displayName", "Route Exception in
Route:To[...]"));
assert rx.test(ticket);
```

Choosing the right listener

- Use Morphia (QueryToFilterListener) when:
 - You are filtering MongoDB collections and want the DB to do the work (indexing, pagination, scalability)
 - You need server-side performance and minimal memory footprint
- Use In-memory (QueryToPredicateJsonListener) when:
 - You run in Quarkus native image and want to avoid reflection on POJOs
 - You are writing unit tests or applying rules to transient/aggregated data
 - You need to evaluate a query over already materialized objects without another database round-trip

Both listeners aim to maintain parity with the grammar. If you observe mismatches, please file an issue with the query string, the evaluated data sample, and the expected vs actual results.

32.3.12. Expressing Ontology Constraints in Queries (optional)

The BIAPI query language primarily targets fields on documents. When using the ontology modules, there are two ways to incorporate ontology relationships into queries:

Option A: Variable from an AccessListResolver

- A resolver computes the set of IDs using EdgeDao (e.g., orders related by placedInOrg to the caller's org).
- Use an IN clause over id in your filter string:

```
# idsByPlacedInOrg is published by an AccessListResolver
id:^{idsByPlacedInOrg}
```

Option B: Function-style helper (if you wire one)

- Some applications register a hasEdge(predicate, value) function into the filter translation path.
- Example (conceptual): hasEdge("placedInOrg", \${principal.orgRefName})
- Under the hood, it translates to id IN edgeDao.srcIdsByDst(tenantId, predicate, dst).

Notes

- Ontology is optional; these patterns only apply when enabled and materialization is active.
- Always scope edge lookups by tenantId from RuleContext.
- See [Integrating Ontology](#) for wiring examples.



Policy function hasEdge()

- The hasEdge(predicate, dstIdOrVar) operator is a policy-level function evaluated before query translation. It is not part of the BIAPI grammar parsed

in this document.

- When present in a policy, the repository typically applies it by calling `ListQueryRewriter.rewriteForHasEdge(...)`, which constrains results by the set of IDs that have the specified ontology edge.
- See [Rule language: add hasEdge\(\)](#) for signature, semantics, and a full end-to-end example.

32.4. Relationships and Ontology-aware Queries

32.4.1. Hydrating related data with `expand(path)`

The query language supports a compact, non-SQL directive to request hydration (materialization) of related entities referenced by the root documents.

- Syntax: `expand(path)`
- Path: a dotted path; array segments may use `[*]` to indicate an array of references.
- Variables: segments may be literal field names or `${var}` placeholders that are substituted before planning.
- Examples:

```
# Single reference
expand(customer)

# Parameterized segment (for tenant-specific collections)
expand(${tenantRoot}.sites[*])

# Array of references
expand(items[*].product)

# Nested paths (bounded by first non-reference boundary)
expand(patient.visits[*].diagnoses[*])
```

Semantics

- Presence of `expand(...)` informs the planner to choose an aggregation-based execution plan (e.g., MongoDB `$lookup`) rather than a simple single-collection filter.
- Filters remain expressed with the existing primitives and compose as usual. For example:

```
# Filter roots by status AND hydrate the related customer
expand(customer) && status:active
```

Notes and limits (v1)

- Backend: MongoDB only in this release. The planner selects aggregation mode when `expand(...)`

appears.

- Depth: for mixed arrays/objects, traversal stops at the first non-reference boundary.
- Projection: a concise projection form `fields:[+a,+b,-c]` can be used at the root (see below). Per-expansion projections and related-entity filters are documented in [Query Expansion](#) and may be introduced incrementally.
- Errors: unknown projection paths (when using `fields:[...]`) are hard errors.
- Variables: `${var}` placeholders in expand paths leverage the same substitution map (request variables plus resolver outputs) as other filter expressions.

See also

- [Query Expansion](#) for more examples and planner behavior.
- [Planner and QueryGateway](#) for how execution mode is chosen.

32.4.2. Ontology operator: `hasEdge(predicate, dst)`

When the ontology module is enabled and edges are materialized, you can constrain results by semantic relationships using `hasEdge`.

- Syntax: `hasEdge(predicate, dst)` where
- `predicate` can be an unquoted string, a quoted string, or a `${var}` placeholder supplied by the caller
- `dst` can be a literal id, a variable like `${principalId}`, an `ObjectId`, or a reference literal `@@...`
- Examples:

```
# Orders placed in a specific organization
hasEdge("placedInOrg", ${orgRefName})

# Tickets routed to a region by id
hasEdge("routedToRegion", 507f1f77bcf86cd799439011)
```

Semantics

- `hasEdge` narrows the result set to entities that have the specified ontology edge from the entity (source) to the destination id/value.
- It composes with other filters using `&&`, `||`, and `!!` like any other expression.
- Tenancy: edge resolution is always scoped by tenant/realm per your configuration.
- Variables: both arguments participate in the standard substitution flow, so resolver output such as `${accessibleOrgPredicate}` and `${orgIds}` can drive ontology constraints without manual string building.

See also

- [Rule language: add hasEdge\(\)](#) for a deep dive and policy usage.

- [Ontology integration](#) for repository wiring and examples.

32.4.3. Root projection recap: `fields:[+...,-...]`

To include or exclude fields from the root documents:

- Syntax: `fields:[+f1,+f2,-f3]`
- Include mode: if any `+` is present, only the listed fields are included (with explicit `-` carving out exceptions).
- Exclude mode: if only `-` entries exist, all fields are included except those.
- Default `_id`: preserved unless explicitly specified (`+_id/-_id`).

Examples

```
# Include a few fields from the root and drop one
fields:[+_id,+total,-internalNotes]

# Combine with expansion (planner chooses aggregation mode)
expand(customer) && fields:[+_id,+customer,+total]
```

For per-expansion projections and advanced traversal filters, see [Query Expansion](#).

Chapter 33. Relationship hydration with `expand(path)`

This section documents the new relationship traversal and hydration capability added to the BIAPI query language. It preserves backward compatibility and introduces no SQL-like syntax.

33.1. How it works

- You add one or more `expand(path)` directives to your query string to request hydration of related entities at specific paths.
- The planner inspects the query. If any `expand(...)` directives are present, it selects AGGREGATION mode; otherwise, FILTER mode.
- In FILTER mode, the system converts the query to a Morphia Filter and queries a single MongoDB collection.
- In AGGREGATION mode, the system constructs an aggregation pipeline (with `$lookup`, `$unwind`, etc.) to materialize related documents. Execution may be guarded by a feature flag in the `QueryGatewayResource`.
- Projection can be applied at the root and per expansion using `fields:[+/-]` syntax. Unknown projection paths are hard errors.

See also:

- [Planner and QueryGateway](#) for mode selection and Java APIs
- [QueryGateway REST endpoints](#) to plan or execute queries over HTTP

33.2. What is `expand(path)`?

Use `expand(path)` to hydrate properties that are modeled as `EntityReference` or ontology-annotated relations. The path is dot-based and may include array wildcards `[*]` between segments.

- Single reference: `expand(customer)`
- Array of references: `expand(items[*].product)`
- Nested arrays: `expand(patient.visits[.diagnoses[]])`

Expansion is a query-time operation: the related document(s) are materialized and embedded at the specified path in the result.

33.3. Filtering related entities (inline)

Continue to use your existing filter primitives in the same query string to filter the root collection. Filtering on related entities will be supported via either:

- Inline path blocks: `path { key:value, key2:^[v1,v2] }`

- Or filters inside `expand(...)`: `expand(customer, status:active)`

Note: In v1, parsing of `expand(path)` is enabled; related-entity filters inside `expand` and inline path blocks will be introduced with the planner/compiler work.

33.4. Projection with fields:[+/-]

A unified projection syntax can be used both at the root and per expansion.

- Inclusion mode: `fields:[+_id,+total,+customer.name]`
- Exclusion mode: `fields:[-internalNotes,-secret]`
- Inside `expand`: `expand(customer, fields:[+name,+tier,-ssn])`

Rules: - If any `+` appears → include only those paths, then apply `-` as carve-outs. - If only `-` appears → include all by default and remove listed paths. - `_id` keeps current default behavior unless explicitly set.

33.5. Examples

- Hydrate a single reference and keep a few fields

```
q = "realm:acme && expand(customer, fields:[+name,+tier]) &&
fields:[+_id,+total,+customer.name]"
```

- Hydrate array of product references inside line items, filter to active products, project a couple of fields

```
q = "expand(items[*].product, active:true, fields:[+sku,+title]) &&
fields:[+_id,+items.product.sku]"
```

- Filter by a related product without projecting it (inline block form)

```
q = "items[*].product{ active:true, sku:^(A123,B456) } && fields:[+_id,+total]"
```

33.6. Semantics and limits

- Backward compatible: if no `expand(...)` or related-path filters are present, queries run as they do today (single-collection).
- Depth for nested paths is bounded; on mixed array/object hops, traversal stops at the first non-reference boundary.
- Unknown projection paths are hard errors (the request fails with a clear message).

See also: [Planner and QueryGateway](#)

33.7. Aggregation examples: what expand(...) compiles to

This section shows typical MongoDB aggregation stages the planner emits when `expand(path)` is present. Exact collection names and field names depend on your model mapping. If the collection cannot be resolved at plan-time, the planner will emit a placeholder from: *"unknown"*; the REST gateway returns HTTP 422 on execution in that case.

Tip: You can inspect the chosen mode and expand paths with `POST /api/query/plan`. See [QueryGateway REST endpoints](#).

33.7.1. Single reference: expand(customer)

Query

```
q = "status:active && expand(customer, fields:[+name,+tier]) &&
fields:[+_id,+total,+customer]"
```

Typical pipeline skeleton

```
[
  { "$match": { "status": "active" } },
  { "$lookup": {
    "from": "customers",
    "localField": "customerId",
    "foreignField": "_id",
    "as": "__exp_customer"
  } },
  { "$unwind": { "path": "$__exp_customer", "preserveNullAndEmptyArrays": true } },
  { "$set": { "customer": "$__exp_customer" } },
  { "$project": { "_id": 1, "total": 1, "customer": { "name": 1, "tier": 1 } } }
]
```

Result shape (conceptual)

```
{
  "_id": "0123",
  "total": 99.50,
  "customer": { "_id": "C1", "name": "Acme", "tier": "GOLD" }
}
```

Notes - The intermediate field `__exp_customer` is an internal staging alias; the planner may omit it in future. - Per-expansion projection `fields:[+name,+tier]` narrows the embedded document via `$project`.

33.7.2. Array of references: expand(items[*].product)

Query

```
q = "expand(items[*].product, fields:[+sku,+title]) && fields:[+_id,+items.product]"
```

Typical pipeline patterns (two common strategies)

1) Unwind-and-regroup (easy to read)

```
[
  { "$unwind": { "path": "$items", "preserveNullAndEmptyArrays": true } },
  { "$lookup": {
    "from": "products",
    "localField": "items.productId",
    "foreignField": "_id",
    "as": "__exp_product"
  } },
  { "$unwind": { "path": "$__exp_product", "preserveNullAndEmptyArrays": true } },
  { "$set": { "items.product": "$__exp_product" } },
  { "$group": {
    "_id": "$_id",
    "doc": { "$first": "$$ROOT" },
    "items": { "$push": "$items" }
  } },
  { "$replaceRoot": { "newRoot": { "$mergeObjects": [ "$doc", { "items": "$items" } ] } } },
  { "$project": { "_id": 1, "items.product": { "sku": 1, "title": 1 } } }
]
```

2) Map-with-lookup (keeps array order without grouping; requires \$lookup with pipeline + \$map)

```
[
  { "$set": {
    "items": {
      "$map": {
        "input": "$items",
        "as": "it",
        "in": {
          "$let": {
            "vars": {
              "p": { "$first": {
                "$lookup": {
                  "from": "products",
                  "let": { "pid": "$$it.productId" },
                  "pipeline": [ { "$match": { "$expr": { "$eq": [ "$_id", "$$pid" ] } } } ],
                  "as": "p"
                }
              }
            }
          }
        }
      }
    }
  } } ],
```

```

        }
      }
    },
    "in": { "$mergeObjects": [ "$$it", { "product": "$p" } ] }
  }
}
}
}
},
{ "$project": { "_id": 1, "items.product": { "sku": 1, "title": 1 } } }
]

```

Notes - The actual compiler may choose either strategy based on simplicity; behavior is equivalent for most cases. - Null or missing productId yields product: null due to preserveNullAndEmptyArrays.

33.7.3. Nested arrays: expand(patient.visits[].diagnoses[])

Query

```

q = "expand(patient.visits[*].diagnoses[*]) &&
fields:[+_id,+patient.visits.diagnoses]"

```

Typical staged pipeline (high level)

1. \$lookup patient -> __exp_patient; \$set patient
2. \$unwind visits (preserveNullAndEmptyArrays)
3. \$unwind visits.diagnoses
4. \$lookup diagnosis -> set visits.diagnoses[*]
5. \$group back by root _id while reconstructing visits and diagnoses arrays in original order
6. optional \$project to constrain fields

Caveats - Depth and mixed object/array hops are bounded; traversal stops at a non-reference boundary. - For very large arrays, prefer targeted filtering before expansion to reduce fan-out.

33.7.4. When is AGGREGATION used vs FILTER?

- FILTER: no expand(...). Planner returns a Morphia Filter and the gateway executes find with dev.morphia.query.Query.
- AGGREGATION: one or more expand(...). Planner returns a pipeline with \$lookup/\$unwind and optional \$project stages. Execution requires feature.queryGateway.execution.enabled=true; otherwise, POST /api/query/find returns 501. If any \$lookup.from is "unknown", execution returns HTTP 422. == Planner and QueryGateway

The planner decides whether a query runs as a simple single-collection filter (FILTER mode) or as a MongoDB aggregation (AGGREGATION mode). This is transparent to callers.

In v1 the planner recognizes `expand(path)` and selects `AGGREGATION` mode when present. Otherwise, `FILTER` mode is used and the existing Morphia path is taken.

33.8. Planner decisions

- `FILTER` mode: No `expand(...)` in the query. The system generates a Morphia Filter and queries a single collection.
- `AGGREGATION` mode: One or more `expand(...)` directives are present. The system will build an aggregation pipeline (with `$lookup`, etc.). In v1, a placeholder pipeline is returned to establish the contract.

33.9. Public facade: QueryGateway

A new collection-agnostic facade exposes plan/inspect capabilities and will later execute aggregation pipelines when enabled.

Java

```
import com.e2eq.framework.model.persistent.morphia.planner.PlannerResult;
import com.e2eq.framework.model.persistent.morphia.query.QueryGateway;
import com.e2eq.framework.model.persistent.morphia.query.QueryGatewayImpl;

public class PlannerSnippet1 {
    static class Order {}
    public static void main(String[] args) {
        QueryGateway gateway = new QueryGatewayImpl();
        PlannerResult pr = gateway.plan("expand(customer) && status:active", Order.class);
        System.out.println(pr.getMode());           // AGGREGATION
        System.out.println(pr.getExpandPaths());    // [customer]
    }
}
```

If you prefer a lower-level API:

Java

```
import com.e2eq.framework.model.persistent.morphia.MorphiaUtils;
import com.e2eq.framework.model.persistent.morphia.planner.PlannedQuery;

PlannedQuery planned = MorphiaUtils.convertToPlannedQuery(
    "expand(items[*].product) && status:active",
    Order.class
);

switch (planned.getMode()) {
    case FILTER:
        // Use planned.getFilter() with Morphia datastore
        break;
    case AGGREGATION:
```

```
// Use planned.getAggregation() to inspect the pipeline (placeholder in v1)
break;
}
```

33.10. Examples end-to-end

- No expansion (FILTER mode):

```
q = "realm:acme && status:active && fields:[+_id,+total]"
```

- With expansion (AGGREGATION mode):

```
q = "expand(customer, fields:[+name,+tier]) && status:active &&
fields:[+_id,+customer.name]"
```

See also: [Relationship hydration with expand\(path\)](#). For concrete pipelines, see [Aggregation examples: what expand\(...\) compiles to](#).

33.11. REST endpoints (v1)

Two helper endpoints expose the planner and execution path. FILTER mode executes immediately. AGGREGATION mode is planned, and execution is guarded by a feature flag.

- POST /api/query/plan — returns planner mode and expand paths detected
- POST /api/query/find — executes FILTER-mode queries and returns the standard Collection<T> envelope; when AGGREGATION is requested, behavior depends on a feature flag (see below)

33.11.1. Request/Response: /api/query/plan

Request body

```
{
  "rootType": "com.e2eq.framework.model.security.UserProfile",
  "query": "expand(customer) && status:active"
}
```

Response body

```
{
  "mode": "AGGREGATION",
  "expandPaths": ["customer"]
}
```

Notes

- mode is FILTER when no expand(...) directives are present; otherwise AGGREGATION.
- expandPaths lists the paths detected by the planner for hydration. See [Query Expansion](#).

33.11.2. Request/Response: /api/query/find

Parameters in the request body

- rootType: fully-qualified class name extending UnversionedBaseModel
- query: BI-API query string (see [Query Language Reference](#))
- page: optional object with limit and skip
- sort: optional array of { field, dir } objects, where dir is ASC or DESC
- realm: optional, overrides the default realm/database

Request body (FILTER mode)

```
{
  "rootType": "com.e2eq.framework.model.security.UserProfile",
  "query": "displayName:*Alice*",
  "page": { "limit": 10, "skip": 0 },
  "sort": [ { "field": "createdDate", "dir": "DESC" } ],
  "realm": "acme"
}
```

Response body (Collection envelope)

```
{
  "offset": 0,
  "limit": 10,
  "rows": [ { "displayName": "Alice T", "refName": "alice" } ],
  "rowCount": 1,
  "filter": "displayName:*Alice*"
}
```

Behavior when AGGREGATION is requested

- If the planner selects AGGREGATION (due to expand(...)) and the configuration property feature.queryGateway.execution.enabled=false (default), the endpoint responds with HTTP 501 Not Implemented and a message indicating execution is disabled.
- If feature.queryGateway.execution.enabled=true, the gateway attempts to execute the aggregation pipeline. If the pipeline contains an unresolved \$lookup.from ("unknown"), the endpoint responds with HTTP 422 Unprocessable Entity and an explanatory error. Otherwise, results are returned in the same Collection envelope shape, with rows being Documents.

cURL examples

Planning with expansion

```
curl -sS -X POST \  
  -H 'Content-Type: application/json' \  
  localhost:8080/api/query/plan \  
  -d '{  
    "rootType": "com.e2eq.framework.model.security.UserProfile",  
    "query": "expand(customer) && status:active"  
  }'
```

Finding with paging, sort, realm (FILTER path)

```
curl -sS -X POST \  
  -H 'Content-Type: application/json' \  
  localhost:8080/api/query/find \  
  -d '{  
    "rootType": "com.e2eq.framework.model.security.UserProfile",  
    "query": "displayName:*Alice*",  
    "page": { "limit": 10, "skip": 0 },  
    "sort": [ { "field": "createdDate", "dir": "DESC" } ],  
    "realm": "acme"  
  }'
```

See also: [Relationship hydration with expand\(path\)](#).

Chapter 34. Security

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 35. 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 36. 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 37. 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 38. 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 39. 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 40. 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 41. 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 42. 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 43. 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 44. 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, SecurityURIHeader, and SecurityURIBody

This section explains how Quantum evaluates permissions for REST requests using rule bases that match on a SecurityURI composed of a header and a body. The header includes identity, area, functionalDomain, and action. The body includes realm, accountNumber, tenantId, dataSegment, ownerId, and resourceId. 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.

Chapter 45. 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 SecurityURI (fine-grained, dynamic)

- What: Declarative rule bases authored against a SecurityURI composed of header (identity, area, functionalDomain, action) and body (realm, accountNumber, tenantId, dataSegment, ownerId, resourceId). Rules decide ALLOW or DENY and may use an optional postconditionScript for additional checks. 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-based matching and simple attribute scoping via the SecurityURI body.
- 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. In the rule engine, roles are evaluated by treating each role as an identity: rules authored for a role name (for example, "ADMIN") are considered alongside rules authored for the user's own userId. 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

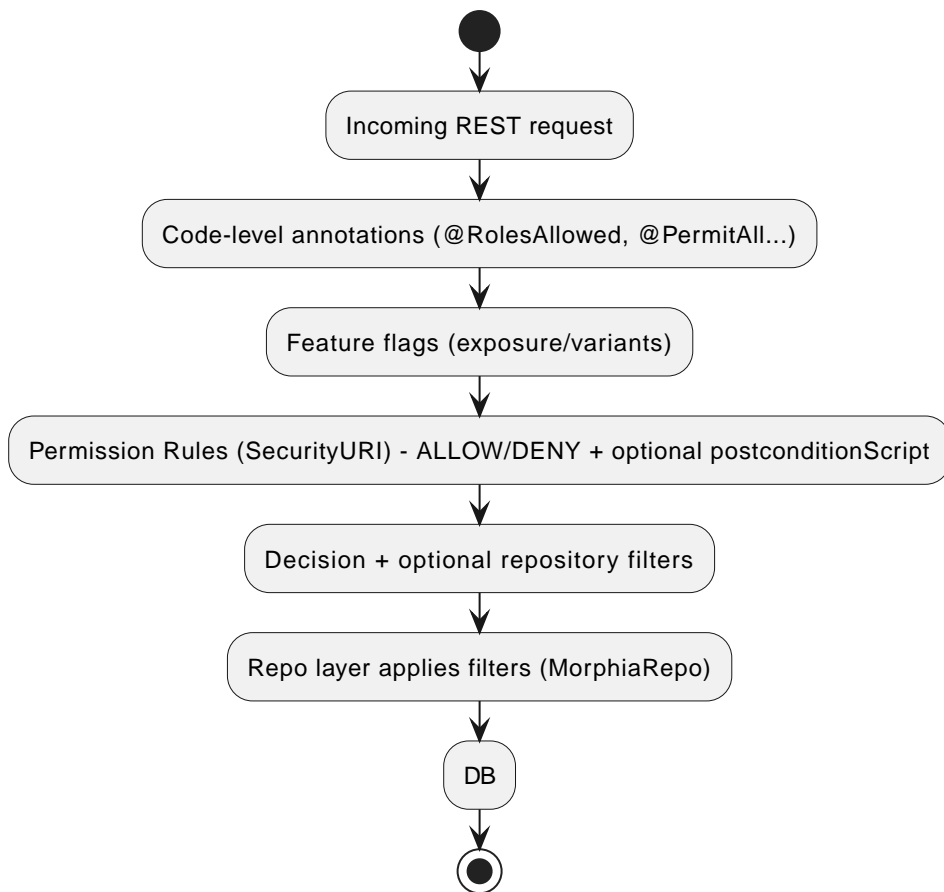
- Use annotations for stable, coarse gates you want visible in code (for example, admin-only endpoints, health endpoints with `@PermitAll`).
- Use feature flags to manage rollout/exposure and variants across environments and cohorts.
- Use permission rules to encode fine-grained, data-aware authorization 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.



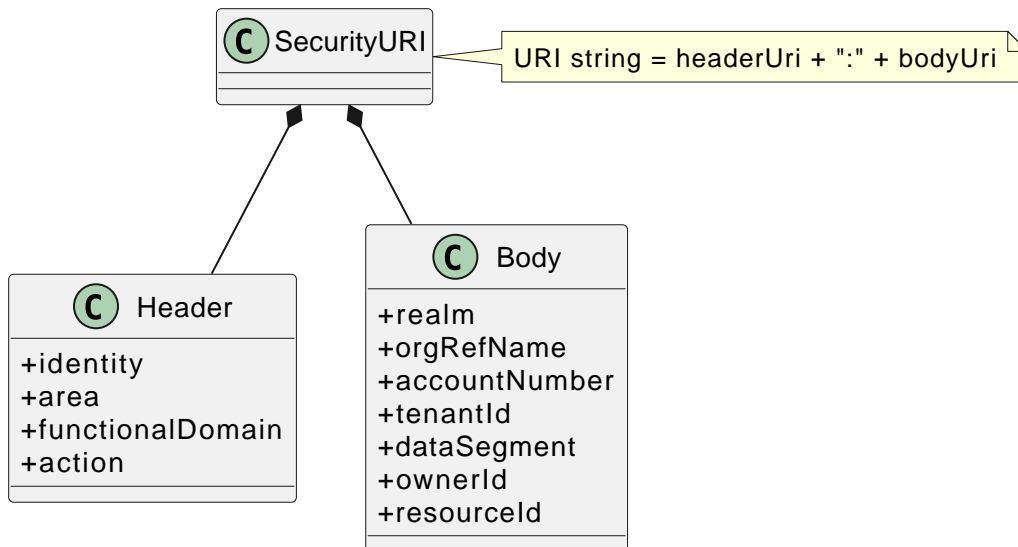
Ontology- and label-aware policy helpers are available to `postconditionScript` when the optional ontology module is enabled. Helpers include `hasEdge`, `hasAnyEdge`, `hasAllEdges`, `relatedIds` for graph checks; `hasLabel` for label checks; and `isA/noViolations` for type/validation contexts. See the "Script Helpers reference" section for details. When data access must be restricted to lists resolved outside the rule engine (for example, from an external ACL service), use `AccessListResolvers` (SPI) and reference their outputs from `andFilterString/orFilterString`.



Chapter 46. 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 are authored against `SecurityURI` and may include:
 - `SecurityURIHeader` fields: identity (`userId` or role name), area, `functionalDomain`, action
 - `SecurityURIBody` fields: realm, `accountNumber`, `tenantId`, `dataSegment`, `ownerId`, `resourceId`
 - Optional `postconditionScript` evaluated with `pcontext/rcontext` for additional checks
 - Priority: integer used to sort rule evaluation (lower numbers evaluated first)
 - Effect: ALLOW or DENY; ALLOWs may be paired with repository-level scoping using the `SecurityURI` body (e.g., `DataDomain` constraints)

Chapter 47. Rule Structure (Illustrative)



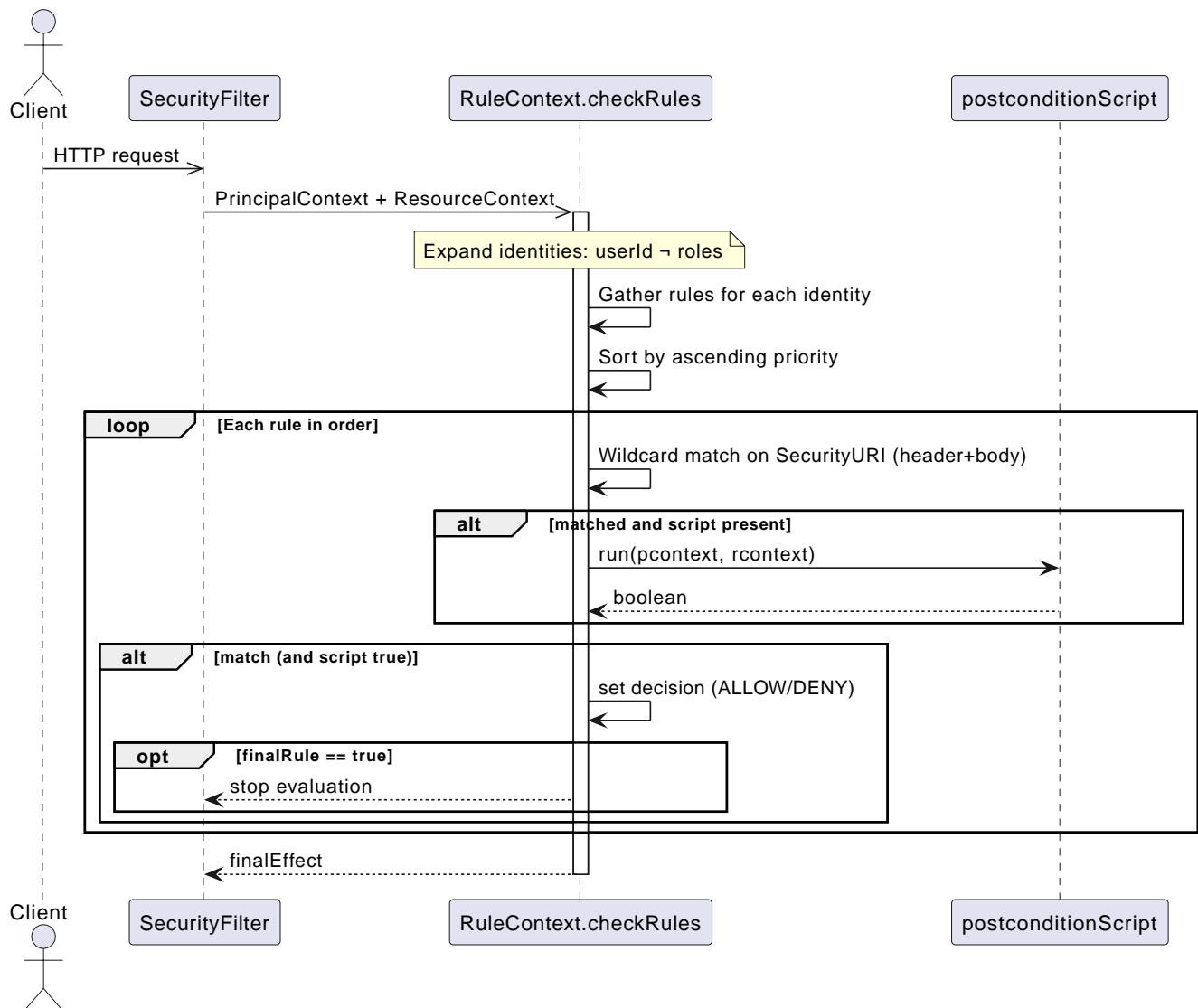
```
- name: allow-catalog-product-reads
description: Allow USER and ADMIN to view products in the Catalog area
securityURI:
  header:
    identity: USER          # or a specific userId; roles are treated as
identities
    area: Catalog
    functionalDomain: Product
    action: view
  body:
    realm: system-com
    accountNumber: '*'
    tenantId: '*'
    dataSegment: '*'
    ownerId: '*'
    resourceId: '*'
  postconditionScript:
  effect: ALLOW
  priority: 300
  finalRule: false

- name: default-deny
description: Fallback deny when nothing else matches
securityURI:
  header:
    identity: '*'
    area: '*'
    functionalDomain: '*'
    action: '*'
  body:
    realm: '*'
    accountNumber: '*'
```

```
tenantId: '*'
dataSegment: '*'
ownerId: '*'
resourceId: '*'
effect: DENY
priority: 10000
finalRule: true
```

- The `securityURI.header` section corresponds to `SecurityURIHeader`: identity (userId or role name), area, functionalDomain, and action. Functional area/domain/action are typically derived from the URL convention `/{{area}}/{{functionalDomain}}/{{action}}` by `SecurityFilter`.
- The `securityURI.body` section corresponds to `SecurityURIBody`: realm, accountNumber, tenantId, dataSegment, ownerId, and resourceId. These values are matched using simple string equality with support for the wildcard `'*'`.
- Optional `postconditionScript` may be provided and is executed as JavaScript with `pcontext` and `rcontext` bindings; the rule only applies if the script evaluates to true.

Chapter 48. Matching Algorithm



The engine evaluates permission rules using SecurityURI wildcard matching. At a high level:

1. Build ResourceContext
 - SecurityFilter derives area, functionalDomain, and action from the request path (or REST annotations if present) and sets the ResourceContext.
2. Expand identities
 - Build a set of identities consisting of the caller's userId plus each effective role. Each of these identities is treated as a potential match target for rules.
3. Gather candidate rules
 - For each identity, collect rules authored for that identity. This forms the candidate set. There is no separate HTTP method/URL or rolesAny/rolesAll matching.
4. Sort by priority
 - Order candidates by ascending priority (lower numbers are evaluated first).
5. URI wildcard comparison

- For each rule in priority order, compare the caller's expanded SecurityURIs to the rule's securityURI using case-insensitive wildcard comparison on the full URI string (header + body). Asterisks (*) in rules match any value.

6. Postcondition script (optional)

- If the rule specifies postconditionScript, execute it as JavaScript with pcontext (principal) and rcontext (resource) variables bound. The rule applies only if the script returns true.

7. Apply effect and finalRule

- When a rule matches (and any script returns true), set the response's finalEffect to the rule's effect (ALLOW or DENY). If finalRule is true, stop evaluating further rules; otherwise continue.

8. Default decision

- If no rule determines a decision, the system returns the default final effect configured by the caller of the check (typically DENY).

Chapter 49. 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 50. 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
  securityURI:
    header: { identity: '*', area: '*', functionalDomain: '*', action: '*' }
    body:   { realm: '*', accountNumber: '*', tenantId: '*', dataSegment: '*',
ownerId: '*', resourceId: '*' }
  effect: DENY
  finalRule: true
```

- Deny-based (use with caution):

```
- name: default-allow
  priority: 10000
  securityURI:
    header: { identity: '*', area: '*', functionalDomain: '*', action: '*' }
    body:   { realm: '*', accountNumber: '*', tenantId: '*', dataSegment: '*',
ownerId: '*', resourceId: '*' }
  effect: ALLOW
  finalRule: true
```

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 51. 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 (SecurityURI 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
  description: Allow ADMIN and REPORTER identities to view export when feature flag is
on
  securityURI:
    header:
      identity: ADMIN      # treat roles as identities
      area: Reports
      functionalDomain: Export
      action: view
    body:
      realm: system-com
      accountNumber: '*'
      tenantId: '*'
      dataSegment: '*'
      ownerId: '*'
      resourceId: '*'
  postconditionScript: userProfile?.features?.EXPORT_API === true
  effect: ALLOW
  priority: 300
  finalRule: false

- name: allow-export-when-flag-on-reporter
  description: Same as above but for REPORTER role
  securityURI:
    header:
      identity: REPORTER
      area: Reports
      functionalDomain: Export
      action: view

```

```

body:
  realm: system-com
  accountNumber: '*'
  tenantId: '*'
  dataSegment: '*'
  ownerId: '*'
  resourceId: '*'
postconditionScript: userProfile?.features?.EXPORT_API === true
effect: ALLOW
priority: 300
finalRule: false

```

Alternatively, systems may surface feature decisions via headers (e.g., X-Feature-SEARCH_V2: v2) so that rules or postconditionScript can read them directly from the request context.

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 52. Multiple Matching RuleBases

- Ordering: rules are evaluated in ascending priority (lower numbers first).
- Decision: when a rule matches, its effect (ALLOW or DENY) becomes the current decision.
- finalRule: if the matching rule has finalRule: true, evaluation stops immediately; otherwise, evaluation continues and a later rule may overwrite the decision.
- Default: if no rule matches decisively, the default effect supplied by the caller (typically DENY) is returned.

Chapter 53. Identity and Role Matching

- Roles-as-identities: the engine evaluates rules for the caller's `userId` and for each effective role by treating each role name as an identity.
- There are no `rolesAny`/`rolesAll` fields in rules; author separate rules for specific roles as needed.
- Optional `postconditionScript` can inspect attributes (for example, `tenantId`) via `pcontext` and `rcontext` when additional checks are needed.
- Time or plan-based conditions can be implemented inside `postconditionScript` or via feature flags.

53.1. 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 54. 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 55. Operational Tips

- Author specific DENY rules with low numbers to prevent accidental exposure.
- Author SecurityURI header (area/domain/action) as narrowly as needed for sensitive domains.
- Prefer SecurityURI body fields and postconditionScript to refine matches without over-broad area/domain/action patterns.
- Log matched rule names and applied scopes for auditability.

Chapter 56. 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, SecurityURI body fields, 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 57. How This Integrates End-to-End

- BaseResource extracts identity and headers to construct DomainContext.
- Rule evaluation uses SecurityURI (header + body) matching with optional postconditionScript and 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 58. 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.

58.1. 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):

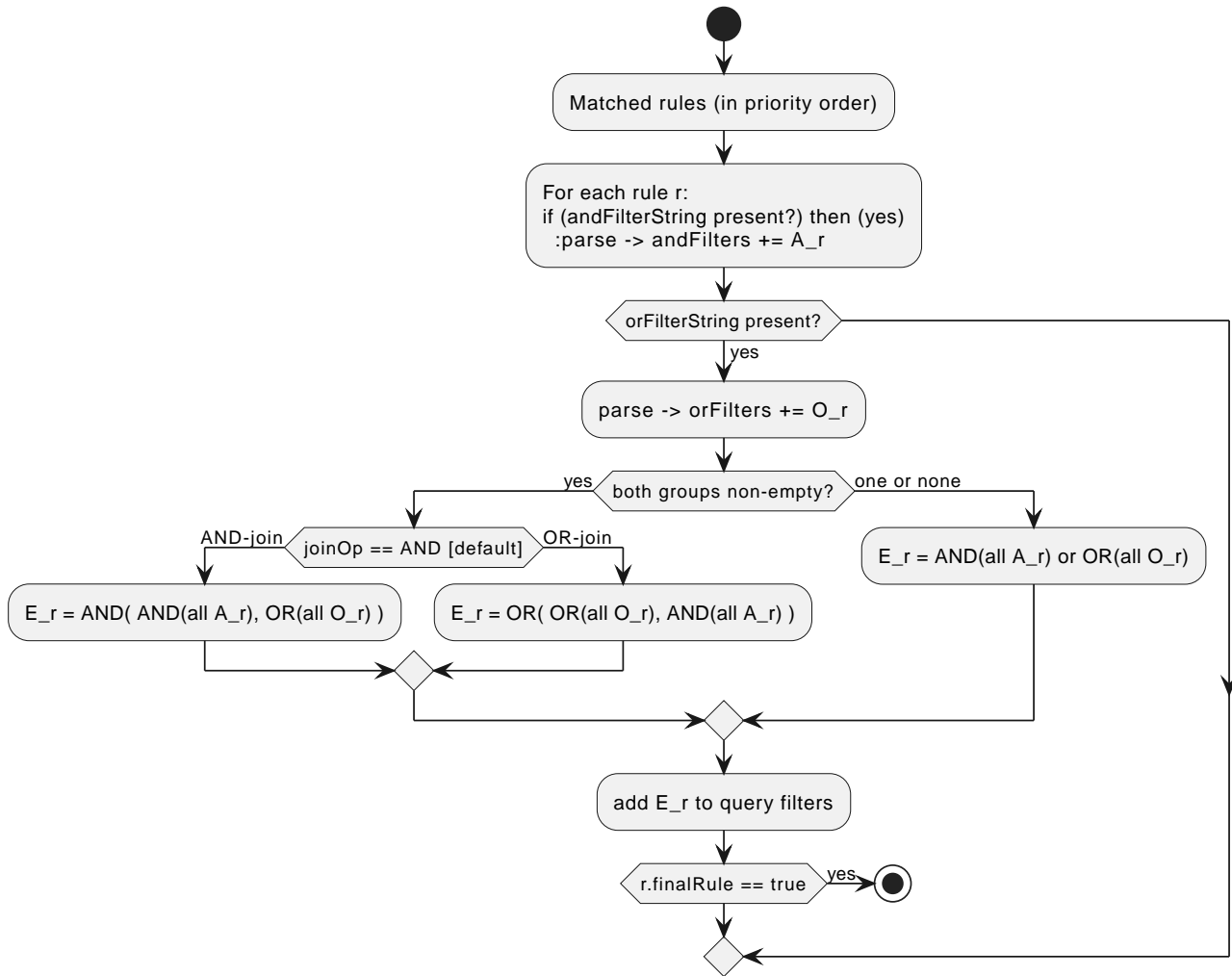
- name, description
- securityURI.header: identity, area, functionalDomain, action (supports wildcard "**")
- securityURI.body: realm, orgRefName, accountNumber, tenantId, ownerId, dataSegment, resourceId (supports wildcard "**")
- effect: ALLOW or DENY
- priority: integer; lower numbers evaluated first
- finalRule: boolean; stop evaluating when this rule applies
- andFilterString / orFilterString: ANTLR filter DSL snippets injected into repository queries (see Query Language section)
- joinOp: how to combine the andFilterString group with the orFilterString group when both are present; defaults to AND

58.2. How rule-contributed filters work (andFilterString, orFilterString, joinOp)

When an ALLOW rule matches, it can contribute repository-level filters that restrict which documents are visible or mutable. The fields are:

- andFilterString: a filter expression added to the AND group.

- `orFilterString`: a filter expression added to the OR group.
- `joinOp`: when both groups are present, specifies how to join them. Values: AND or OR. Default: AND.



Composition algorithm (implemented in `RuleContext.getFilters`):

1. Collect all matched rules (in priority order; skipping NOT_APPLICABLE script results). For each rule:
 - If `andFilterString` is set, parse it into a Morphia Filter and add to the and-group.
 - If `orFilterString` is set, parse it into a Morphia Filter and add to the or-group.
2. If both groups are non-empty for the current rule:
 - If `joinOp == AND` (default): `effective = AND(AND(all and-group), OR(all or-group))`.
 - If `joinOp == OR`: `effective = OR(OR(all or-group), AND(all and-group))`.
3. If only one group is non-empty:
 - Use `AND(all and-group)` or `OR(all or-group)` as the effective filter.
4. Append the effective filter(s) to the query's filter list. If the rule has `finalRule: true`, stop accumulating more filters.
5. Deduplicate filters by string form before returning.

Examples:

- AND only

```
- name: tenant-scope
  securityURI:
    header: { identity: USER, area: sales, functionalDomain: order, action: view }
    body: { realm: '*', orgRefName: '*', accountNumber: '*', tenantId: '*', ownerId:
'*', dataSegment: '*', resourceId: '*' }
    andFilterString: "dataDomain.tenantId:${pcontext.dataDomain.tenantId}"
    effect: ALLOW
    priority: 200
```

Resulting Morphia: AND(eq("dataDomain.tenantId", <caller-tenant>))

- OR only

```
- name: visibility-by-segment
  securityURI:
    header: { identity: USER, area: sales, functionalDomain: order, action: view }
    body: { realm: '*', orgRefName: '*', accountNumber: '*', tenantId: '*', ownerId:
'*', dataSegment: '*', resourceId: '*' }
    orFilterString: "dataDomain.dataSegment:^(PUBLIC|INTERNAL)"
    effect: ALLOW
    priority: 210
```

Resulting Morphia: OR(in("dataDomain.dataSegment", [PUBLIC, INTERNAL]))

- AND + OR with joinOp: AND (default)

```
- name: own-or-public
  securityURI:
    header: { identity: USER, area: security, functionalDomain: userProfile, action:
view }
    body: { realm: '*', orgRefName: '*', accountNumber: '*', tenantId: '*', ownerId:
'*', dataSegment: '*', resourceId: '*' }
    andFilterString: "dataDomain.ownerId:${principalId}"
    orFilterString: "dataDomain.dataSegment:^(PUBLIC)"
    joinOp: AND
    effect: ALLOW
    priority: 220
```

Effective: AND(eq(ownerId, principalId), OR(eq(dataSegment, 'PUBLIC')))

- AND + OR with joinOp: OR

```
- name: owner-or-segment
```

```

securityURI:
  header: { identity: USER, area: files, functionalDomain: document, action: view }
  body:   { realm: '*', orgRefName: '*', accountNumber: '*', tenantId: '*', ownerId:
'*', dataSegment: '*', resourceId: '*' }
  andFilterString: "dataDomain.ownerId:${principalId}"
  orFilterString:  "tags:^[shared]"
  joinOp: OR
  effect: ALLOW
  priority: 230

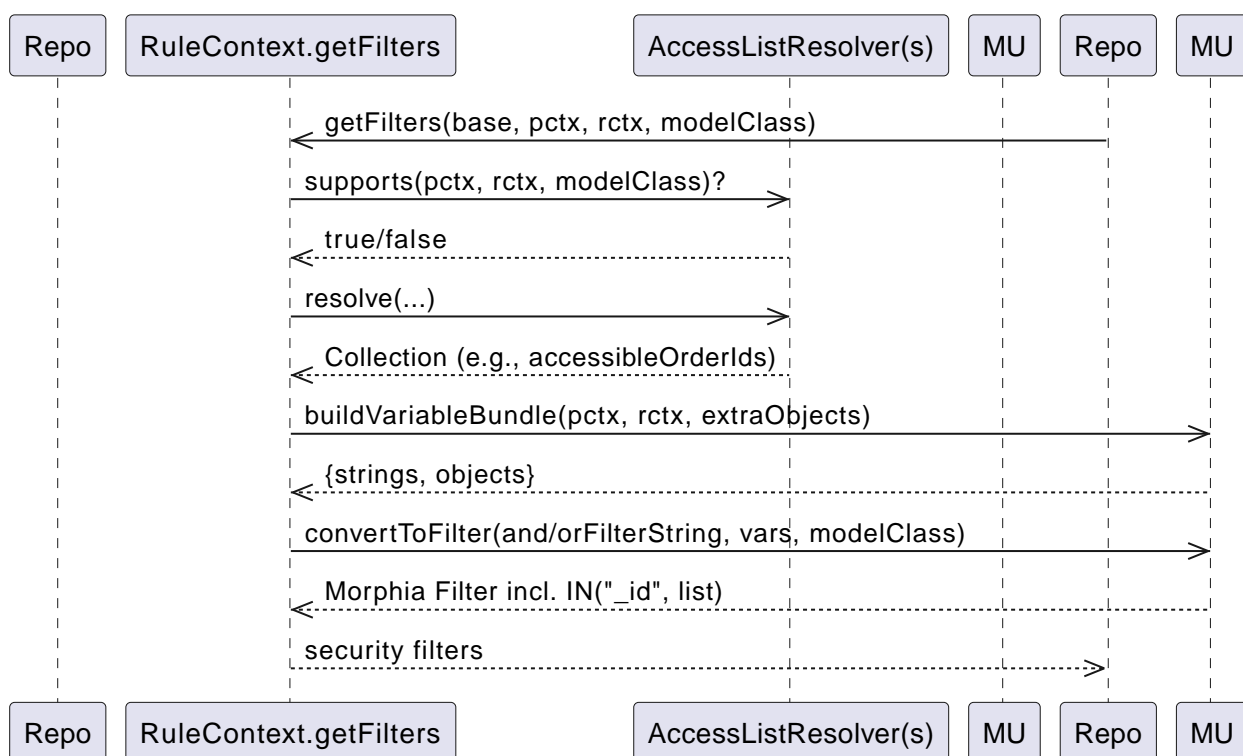
```

Effective: OR(eq(ownerId, principalId), AND(eq(tags, 'shared')))

Placeholders and variables:

- `${principalId}`, `${pTenantId}`, and other variables are resolved from `PrincipalContext/ResourceContext` and `AccessListResolvers`.
- The filter DSL is described in the Query Language guide; it maps to Morphia `dev.morphia.query.filters.Filters` under the hood via `MorphiaUtils.convertToFilter`.

58.3. AccessListResolvers (SPI) for list-based access



`AccessListResolvers` let you plug in computed collections (IDs, codes, emails, etc.) at request time and reference them from `andFilterString/orFilterString`. This is ideal for ACL-style list checks or integrating with external systems that decide which resources a user may access.

How it works (as implemented):

- SPI: `com.e2eq.framework.securityrules.AccessListResolver`
- `key()`: the variable name published to the filter variable bundle (e.g., `"accessibleCustomerIds"`).

- supports(pctx, rctx, modelClass): return true when this resolver applies for the current request and repository model type.
- resolve(pctx, rctx, modelClass): return a Collection<?> which will be available as \${<key>} in filter strings.
- RuleContext.getFilters discovers all AccessListResolver beans via CDI, calls supports(...), and for those that apply it puts key() → resolve(...) into the "extraObjects" map. MorphiaUtils.buildVariableBundle merges these into the variable set used by convertToFilter.
- In your filter DSL, use:
- IN with resolver-provided lists: field: ^\${key}
- Equality with resolver-provided scalars: field: \${key}
- You can also reference nested principals: \${pcontext.dataDomain.tenantId}, \${principalId}, etc.

Examples:

- Restrict Orders to the set of ids returned by a resolver

```
- name: orders-by-acl
  securityURI:
    header: { identity: USER, area: sales, functionalDomain: order, action: view }
    body:   { realm: '*', orgRefName: '*', accountNumber: '*', tenantId: '*', ownerId:
'*', dataSegment: '*', resourceId: '*' }
    andFilterString: "_id: ^${accessibleOrderIds}"
    effect: ALLOW
    priority: 200
```

- Use resolver that returns customer codes and OR with a public segment

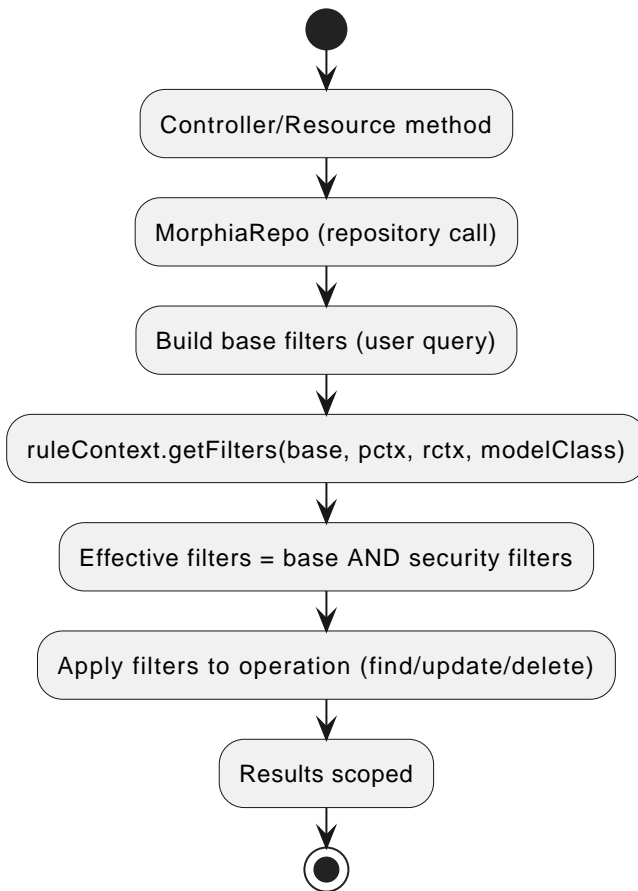
```
- name: customer-code-or-public
  securityURI:
    header: { identity: USER, area: crm, functionalDomain: customer, action: view }
    body:   { realm: '*', orgRefName: '*', accountNumber: '*', tenantId: '*', ownerId:
'*', dataSegment: '*', resourceId: '*' }
    andFilterString: "code: ^${visibleCustomerCodes}"
    orFilterString:  "dataDomain.dataSegment: ^[PUBLIC]"
    joinOp: OR
    effect: ALLOW
    priority: 210
```

Reference implementation (tests/examples): - RuleContext.getFilters collects resolvers and exposes them to the filter StringSubstitutor via MorphiaUtils.buildVariableBundle. - See also: quantum-framework/src/test/java/com/e2eq/framework/securityrules/TestCustomerAccessResolver.java and TestStringAccessResolver.java for sample resolvers.

Guidance: - Keep keys stable and document them; they form your contract between resolver authors and rule authors. - Always scope resolver queries by tenant/realm from

PrincipalContext/ResourceContext. - Return empty collections instead of null. An empty IN list yields no matches, which is safe by default.

Where filters are applied (MorphiaRepo methods):



- `getList` / `getListByQuery` / `find`: `RuleContext.getFilters` augments the base filters with the effective security filters before executing the query.
- `count`: same as read paths, ensuring counts reflect scoped visibility.
- `save (create)`: rules typically do not inject filters for inserts; writes are validated by separate preconditions or `postconditionScript`. If your policy encodes write-scope, author `UPDATE/DELETE` rules to guard modifications rather than `CREATE`, unless your domain enforces ownership/tenant on insert.
- `update` / `merge` / `set` / `bulk set`: the security filters are ANDed into the target selection so only documents visible under the policy are affected.
- `delete (by id or by query)`: security filters are applied to the selection to prevent deleting outside the allowed scope.

Implementation reference:

```
// Repositories call into RuleContext to augment filters
List<Filter> filters = ruleContext.getFilters(baseFilters,
    SecurityContext.getPrincipalContext().get(),
    SecurityContext.getResourceContext().get(),
    getPersistentClass());
```

```
Query<T> query = datastore.find(getPersistentClass()).filter(filters.toArray(new
Filter[0]));
```

Notes:

- Filters are applied regardless of whether the match decision was ALLOW or DENY; only ALLOW rules contribute filters. DENY rules decide the outcome but do not add filters.
- finalRule: true stops evaluating later rules for both decision and filter contribution.
- If no ALLOW rules match, repositories may still execute with caller-provided filters, but upstream permission checks should have DENIED the action; by convention, most endpoints call checkRules and short-circuit DENY before hitting the database.

Example impact per MorphiaRepo method:

- save: no security filters are injected into the insert operation; enforce ownership/tenant fields via model validation and/or postconditionScript.
- find/findById: adds security filters; if the requested id is outside scope, the result is empty.
- getList/getListByQuery: adds security filters to user-supplied query; scope cannot be broadened by the client.
- update/merge: AND the security filters into the update selector; records outside scope are unaffected.
- delete: AND the security filters into the delete selector; out-of-scope records are not removed.

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
}
]
}

```

58.4. 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

58.5. 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"
```

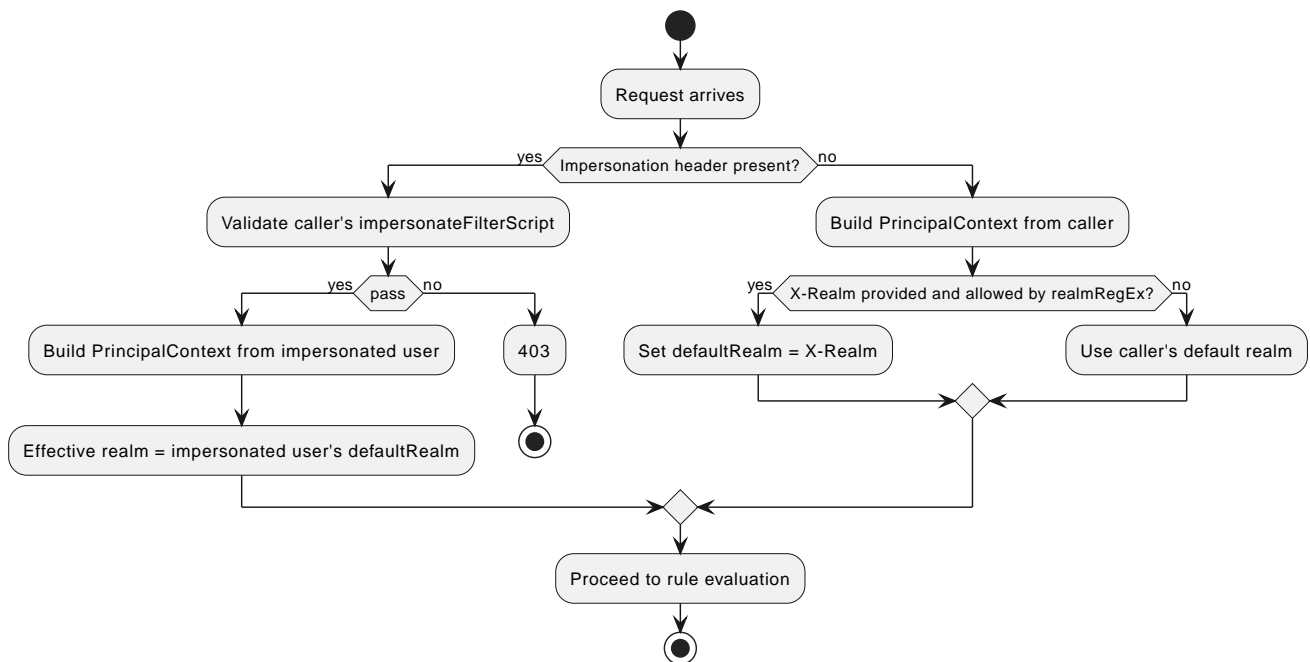
58.6. How changes affect rule bases and enforcement

- 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 59. 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.

59.1. 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.

59.2. 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.

59.3. 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.

59.4. 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.

59.5. 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.

59.6. 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 60. 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.

60.1. 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.

60.2. 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.

60.3. 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`.

60.4. 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.

60.5. 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

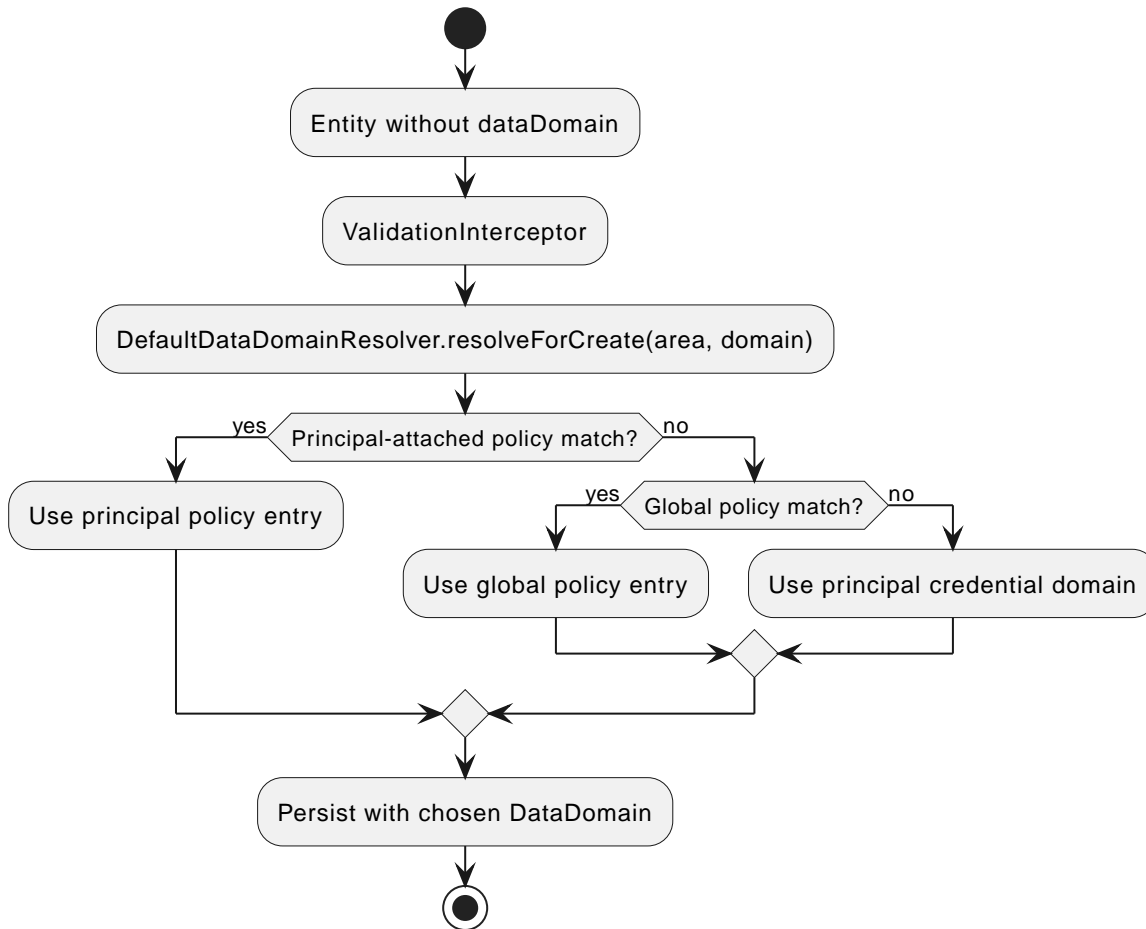
Each `DataDomainPolicyEntry` has a `resolutionMode`: - `FROM_CREDENTIAL` (default): use the principal's credential domain (i.e., the historical behavior). - `FIXED`: use the first `DataDomain` listed in `dataDomains` on the entry.

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.



60.6. 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.

60.7. 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.

60.8. 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.

60.9. 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.

60.10. 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.

60.11. Ontology in Permission Rules (optional)

If you enable the ontology modules, you can author rules that constrain access by semantic relationships, not field paths. This keeps policies stable as your object model evolves.

Key idea

- Materialize edges in Mongo using `OntologyMaterializer` (e.g., `placedInOrg`, `orderShipsToRegion`).
- During rule evaluation, translate a semantic constraint like "has edge `placedInOrg` to `OrgX`" into a set of IDs, and combine that with your query.

How to use

- Preferred: author a rule with a semantic hint and let the application translate it via `ListQueryRewriter`.
- Minimal change path: publish a variable via an `AccessListResolver` and use an IN filter over `_id`.

Example (resolver + IN filter)

- Add an `AccessListResolver` that returns order IDs for which `(tenantId, p="placedInOrg", dst=orgRefName)` exists.
- In your rule's AND filter string (query language), use: `id: ^${idsByPlacedInOrg}`

See also

- Ontology overview and examples: [Ontologies in Quantum](#)
- Integration with Morphia and multi-tenancy: [Integrating Ontology](#)

Operational notes

- Ontology is optional. Enable it per service when the config flag and dependencies are present.
- Always scope edge queries by `tenantId` sourced from `RuleContext`.
- Index edges on `(tenantId, p, dst)` and `(tenantId, src, p)` for performance.

60.11.1. Rule language: add `hasEdge()`

We introduce a policy function/operator to reference ontology edges directly from rules. This lets policies constrain access by semantic relationships instead of field paths.

Signature

- `hasEdge(predicate, dstIdOrVar)`

Parameters

- `predicate`: String name of the ontology predicate (e.g., "placedInOrg", "orderShipsToRegion").
- `dstIdOrVar`: Either a concrete destination id/refName or a variable resolved from RuleContext (e.g., `principal.orgRefName`, `request.region`).

Semantics

- The rule grants/filters entities for which an edge exists: (`tenantId`, `src` = `entity._id`, `p` = `predicate`, `dst` = `resolvedDst`).
- Multi-tenant safety: `tenantId` is always taken from RuleContext/DomainContext.

Composition

- `hasEdge` can be combined with existing rule clauses (and/or/not) and other filters (states, tags, `ownerId`, etc.).

Examples

- Allow viewing Orders in the caller's org (including ancestors via ontology closure):
- allow VIEW Order when `hasEdge("placedInOrg", principal.orgRefName)`
- Restrict list to Orders shipping to a region chosen in request:
- allow LIST Order when `hasEdge("orderShipsToRegion", request.region)`

Under the hood

- Policy evaluation resolves `dstIdOrVar` against RuleContext (for example, `principal.orgRefName` → "OrgP").
- The list/filter query is rewritten using `ListQueryRewriter.rewriteForHasEdge(...)`, which turns the predicate into a set of source ids and merges it with the base query efficiently.
- `OntologyEdgeDao` must be indexed on (`tenantId`, `p`, `dst`) and (`tenantId`, `src`, `p`) for performance.

Full end-to-end example (implementation pattern)

1) Author a rule (illustrative YAML/pseudocode)

```
- name: list-orders-by-org
  priority: 100
  securityURI:
    header:
      identity: USER
      area: sales
      functionalDomain: order
      action: list
```

```

body:
  realm: '*'
  accountNumber: '*'
  tenantId: '*'
  dataSegment: '*'
  ownerId: '*'
  resourceId: '*'
effect: ALLOW
# Semantic constraint expressed via ontology helper in postconditionScript
postconditionScript: hasEdge("placedInOrg", pcontext?.dataDomain?.orgRefName) ===
true

```

2) Evaluate policy and apply constraint in the repository/list path

```

import com.e2eq.ontology.policy.ListQueryRewriter;
import com.e2eq.ontology.repo.OntologyEdgeRepo;
import com.mongodb.client.model.Filters;
import org.bson.conversions.Bson;

// Injected once per service when ontology is enabled (Quarkus CDI)
@jakarta.inject.Inject OntologyEdgeDao edgeDao;
ListQueryRewriter rewriter = new ListQueryRewriter(edgeDao);

// Inside your list method, after building the base filter
String tenantId = ruleContext.getRealmId(pctx, rctx); // or from DomainContext
String predicate = "placedInOrg";
String orgRefName = pctx.getDataDomain().getOrgRefName(); // resolves
principal.orgRefName

Bson base = Filters.and(existingFilters...);
Bson rewritten = rewriter.rewriteForHasEdge(base, tenantId, predicate, orgRefName);
var results = datastore.getDatabase().getCollection("orders").find(rewritten).
iterator();

```

3) Morphia-typed queries alternative

```

// If you're using Morphia's typed Query API
Set<String> ids = edgeDao.srcIdsByDst(tenantId, "placedInOrg", orgRefName);
if (ids.isEmpty()) {
  return java.util.List.of(); // short-circuit
}
query.filter(dev.morphia.query.filters.Filters.in("_id", ids));

```

Developer requirements and checklist

- Enable ontology (optional feature): set `e2eq.ontology.enabled=true` in your app and add module dependencies.
- Provide a TBox (OntologyRegistry) and run `OntologyMaterializer` on entity changes to keep

edges up to date.

- Inject EdgeDao as a CDI bean (@Inject); indexes are ensured automatically at startup.
- Always pass tenantId from RuleContext/DomainContext; never cross tenants.
- When using variables on the RHS (dstIdOrVar), ensure the RuleContext exposes them (for example, principal.orgRefName or request.region).
- Monitor provenance (edge.prov) and re-materialize edges when intermediate nodes change.

Notes

- hasEdge() is a policy function; it is evaluated before constructing database filters. It is not part of the core BIAPI query grammar.
- If ontology is disabled, skip the rewrite (return base) or configure a no-op implementation so policies that include hasEdge are rejected early with a clear error message.

60.12. Label resolution SPI and hasLabel() in rules

Labels are a lightweight way to attach semantic markers to principals and resources, and then use them in rule scripts. Quantum provides a pluggable label resolution SPI and a script helper to check labels during policy evaluation.

Why labels? - Decouple policy from rigid fields (e.g., “VIP”, “HARD_DELETE_DISABLED”, “B2B”). - Compute labels from multiple sources (simple tags, advancedTags, dynamic attributes, or custom derivations).

Components

- LabelResolver (SPI):

```
public interface LabelResolver {
    boolean supports(Class<?> type);
    java.util.Set<String> resolveLabels(Object entity);
}
```

- DefaultLabelResolver (built-in): Applies to UnversionedBaseModel.
- Collects tags (String[]), advancedTags.name, and optionally any dynamicAttributes whose isInheritable()==true, using best-effort reflection.
- LabelService: Aggregates all LabelResolver beans via CDI and delegates to the first that supports the entity type. Also supports annotation-based extraction (see below).
- Annotations (optional, opt-in at your model):

```
@LabelSource(method = "computeLabels") // class-level method returning
Collection<String>, String[], or String
public class MyEntity { ... }

public class MyEntity {
```

```

@LabelField // field can be Collection<String>, String[], or
String
private java.util.List<String> policyLabels;
}

```

Availability in rule scripts

During policy evaluation, RuleContext resolves labels for both principal and resource contexts and installs a script helper:

- hasLabel(label: String) → Boolean
- Returns true if the current resource context has the specified label.
- Label resolution uses LabelService (including custom resolvers and annotations) at evaluation time.

Examples (postcondition script)

```

// Allow if the resource is labeled VIP
hasLabel("VIP")

// Combine with other helpers when ontology is enabled
hasLabel("RESTRICTED") && !hasEdge("placedInOrg", "OrgX")

```

Extending labels for your domain

- Add a new resolver:

```

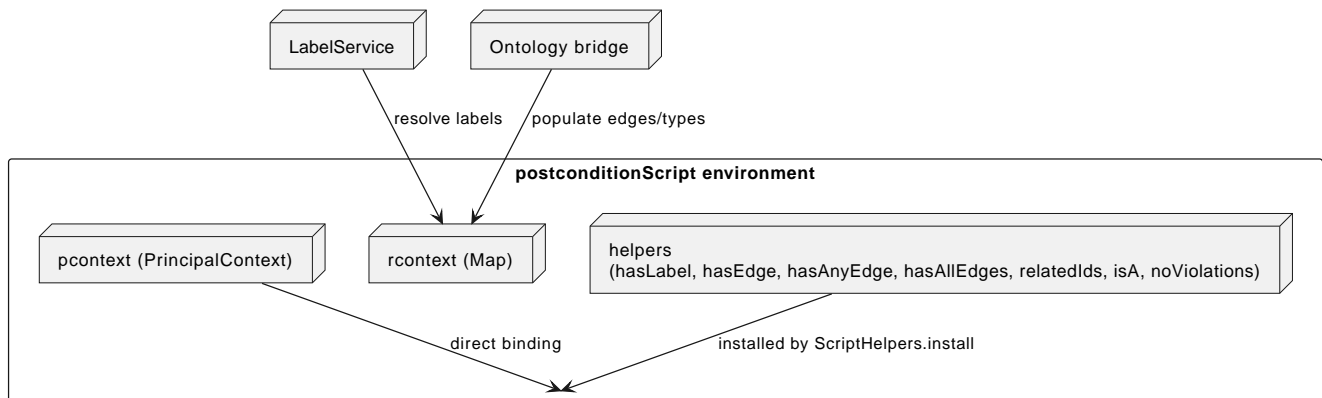
@jakarta.enterprise.context.ApplicationScoped
public class InvoiceLabelResolver implements LabelResolver {
    public boolean supports(Class<?> type) { return type.getName().endsWith("Invoice"); }
    public java.util.Set<String> resolveLabels(Object entity) {
        var out = new java.util.LinkedHashSet<String>();
        var inv = (com.example.Invoice) entity;
        if (inv.isHighValue()) out.add("HIGH_VALUE");
        if (inv.isPastDue()) out.add("PAST_DUE");
        return out;
    }
}

```

- Or annotate your model with @LabelSource/@LabelField to contribute labels without writing a resolver.

Notes - Labels are resolved best-effort; failures are swallowed to keep policy evaluation robust. - If multiple resolvers exist, the first that supports the type wins. Prefer narrow supports() checks. - Keep label names stable; treat them as part of your policy contract.

Chapter 61. Script Helpers reference (when enabled)



Script helpers are small functions injected into the `postconditionScript` environment to make common policy checks concise. They are provided by `com.e2eq.ontology.policy.ScriptHelpers.install(...)` when the optional ontology/label modules are present. In all cases, `postconditionScript` also has direct access to:

- `pcontext`: the Java `PrincipalContext` for the caller
- `rcontext`: a lightweight map for resource-related helper data (labels, types, edges, violations) when populated by your application

Availability notes: - Labels are made available by `RuleContext` via `LabelService`. `hasLabel()` checks the resource's labels only. - Edges, types, and violations must be populated by your application into the `rcontext` map before calling `RuleContext.runScript` (typically via an ontology bridge). If absent, edge/type helpers return `false/empty`.

Helpers and examples:

- `isA(type: String) → Boolean`
- Purpose: Check whether the resource has a semantic type in `rcontext.types`.
- Example:

```
// Allow when the resource is of type "Invoice"
isA("Invoice")
```

- `hasLabel(label: String) → Boolean`
- Purpose: Check whether the resource has a policy label (labels resolved via `LabelService`).
- Example:

```
// Permit view if resource is labeled PUBLIC
hasLabel("PUBLIC")
```

- `hasEdge(predicate: String, dst: String|null) → Boolean`
- Purpose: True if an ontology edge with property `p=predicate` exists from the current resource to `dst`. If `dst` is null, true if any `dst` exists for the predicate.
- Example:

```
// Require that this resource is placed in the caller's org
hasEdge("placedInOrg", pcontext?.dataDomain?.orgRefName)
```

- `hasAnyEdge(predicate: String, dsts: Collection<String>) → Boolean`
- Purpose: True if there is at least one edge to any of the destinations.
- Example:

```
// Allow if the order ships to any of the selected regions
hasAnyEdge("orderShipsToRegion", ["NA", "EU"]) === true
```

- `hasAllEdges(predicate: String, dsts: Collection<String>) → Boolean`
- Purpose: True only if there is an edge for every destination in the list.
- Example:

```
// Require all compliance flags to be present
hasAllEdges("hasComplianceFlag", ["KYC", "AML"]) === true
```

- `relatedIds(predicate: String) → List<String>`
- Purpose: Return the list of `dst` ids for edges with property `p=predicate`.
- Example:

```
// Use in conjunction with a filter list variable (via AccessListResolver)
var projectIds = relatedIds("belongsToProject");
projectIds && projectIds.length > 0
```

- `noViolations() → Boolean`
- Purpose: True if `rcontext.violations` is empty. Helpful when upstream validators populate violations.
- Example:

```
// Deny action if any violations were detected earlier in the pipeline
noViolations() === true
```

Best practices: - Prefer simple boolean expressions; keep `postconditionScript` fast and side-effect free. - When using ontology helpers, always scope your edge materialization by `tenantId` to avoid

cross-tenant leakage. - Combine helpers with SecurityURIBody scoping and repository filters for defense in depth.

Cross-references: - Ontology usage and hasEdge: see [Ontology in Permission Rules](#). - Label SPI and hasLabel: see [Label resolution SPI](#). - AccessListResolver for list variables in filters: see "AccessListResolvers (SPI) for list-based access" above.

61.1. 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.

61.1.1. @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
}
```

61.1.2. @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);
}
}

```

61.1.3. 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
DELETE	DELETE

61.1.4. 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

61.1.5. 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

61.1.6. 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) { ... }
```

61.1.7. 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
```

61.1.8. See Also

- [Modeling: Functional Mapping](#)

- [Permission Rules](#)
- [DomainContext and RuleContext](#)

Chapter 62. Advanced Features

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 63. 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 64. 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 65. 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 66. 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` considers a change set pending if either (a) it has never run, (b) the bean's `changeSetVersion` is greater than the last recorded one, or (c) the bean's checksum is non-null and differs from the last recorded checksum in `ChangeSetRecord`. It also compares each change set's `dbToVersion` against the stored `DatabaseVersion`.
- **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 67. 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`
- `getChangeSetVersion()` [optional]: an integer you bump when the definition/logic of this change set evolves but its target database version does not. Defaults to 1.
- `getChecksum()` [optional]: a stable checksum string representing the content/logic of the change set. If provided and it changes, the change set will re-run even if versions did not change.

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 68. 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 69. 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 70. 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 71. 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 72. 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 73. 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.

73.1. Checksum vs. from/to versions

ChangeSetBean introduces two optional mechanisms that influence whether a change set runs again after it has already executed:

- `changeSetVersion` (int): A manual counter you increment when you intentionally want the same change set (same to-version) to run again because its logic changed or needs to reapply. Defaults to 1.
- `checksum` (String): A content-derived fingerprint of the change set's implementation. If provided, the framework will re-run the change set whenever the checksum differs from the last recorded value, even if versions and `changeSetVersion` are unchanged.

How they complement `getDbFromVersion/getDbToVersion`:

- `getDbFromVersion/getDbToVersion` define the database version boundary the change set moves the realm across. They control ordering and whether the database version should advance on success.
- `changeSetVersion/checksum` control repeatability of a change set at a fixed to-version. They do not change ordering; they only indicate that the change set should be applied again.

Typical usage patterns:

- Minor logic refactor that should re-apply: bump `changeSetVersion`, or update checksum if you compute it from code/resources.
- Data or index definition tweaked without a version bump: provide `getChecksum()` that reflects the relevant definitions (e.g., hash of DDL/index specs or embedded dataset), so the system auto-detects changes and re-runs.
- Actual schema version change: bump `getDbToVersion` (and possibly from) as usual; you may leave `changeSetVersion` and checksum alone.

Recording and behavior in `ChangeSetRecord`:

- On each execution, MigrationService stores changeSetVersion and checksum alongside from/to versions.
- A change set is considered pending if no record exists, or the bean's changeSetVersion is greater than the recorded one, or the bean's checksum is non-null and differs from the recorded checksum.
- After a successful run, DatabaseVersion is updated to dbToVersion if it is higher than current.

Caveats:

- If you return null from getChecksum(), only first-run and changeSetVersion increases can trigger re-execution.
- Ensure your checksum is stable across processes and deterministic for the same logic; do not include timestamps or environment-specific data.
- When migrating large datasets, prefer idempotent logic so repeated runs are safe even if checksum forces a re-run. == Testing in Quantum: Security Contexts, Repos, and REST APIs

73.2. 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.

73.3. 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).

73.4. 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.

73.5. 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 findByUserId_asTestUser() {
        try (final com.e2eq.framework.securityrules.SecuritySession s =
            new com.e2eq.framework.securityrules.SecuritySession(pContext, rContext))
        {
            var op = credRepo.findByUserId(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`.

73.6. 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.

73.7. 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.

73.8. 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).

73.9. 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.

73.10. 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.

73.11. 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 74. Tutorials

Supply Chain Collaboration SaaS: A Business-First Guide

This guide explains, in plain language, how the Quantum framework helps you build a Supply Chain Collaboration Software-as-a-Service (SaaS). We focus on real business problems—secure data sharing, multi-party workflows, and tenant isolation—and show how the framework’s building blocks solve them without requiring you to stitch together dozens of bespoke APIs.

Chapter 75. What a supply-chain SaaS needs (and how Quantum helps)

Common needs when launching a collaboration platform:

- Secure sharing across companies: Buyers, suppliers, carriers, and 3PLs must see the same truth, but only what they're allowed to see.
- Role-appropriate views: Planners, operations, and analysts look at the same orders/shipments but need different fields, filters, and actions.
- Auditability and control: You need a clear explanation of who saw or changed what, and why that was allowed.
- Fast onboarding: Each partner authenticates differently; you can't force everyone into one identity provider.
- API consistency: Your UI and integrations shouldn't learn a different API for every screen or entity.
- Data lifecycle and stewardship: Easy data imports/exports, clear status tracking, and repeatable completion steps.

How Quantum maps to these needs:

- Multi-tenancy by design: Each organization (tenant) is isolated by default; sharing is added deliberately and safely.
- Policy-driven permissions: Human-readable rules answer "who can do what" and add scoping filters so only the right data shows up.
- Consistent REST and Query: One List API and a simple query language cover most searches and reports—no explosion of bespoke endpoints.
- Flexible identity: Support for different authentication methods per organization, plus delegated admin so each tenant manages its own users.
- State + tasks: Built-in patterns for long-running, stateful processes and "completion tasks" to move work forward predictably.

Chapter 76. Why multi-tenancy is a natural fit for supply chains

Supply chains are networks. Everyone shares a process, but not a database. Quantum isolates each tenant's data by default and lets you selectively share records with partners:

- Private by default: A supplier's purchase orders are not visible to other suppliers.
- Share on purpose: Create a "collaboration bubble" around a purchase order or shipment so a buyer and a specific carrier can see the same milestones and documents.
- Central where it helps: Keep a global partner directory or product catalog in a shared domain if that reduces duplication.
- Regional and regulatory needs: Pin certain data (e.g., EU shipments) to the correct region with simple policies.

Chapter 77. Who uses the system (organizations and roles)

Organizations (tenants) - Shippers and customers: create orders, monitor shipments, approve changes. - Carriers and 3PLs: accept tenders, provide status, confirm delivery. - Suppliers: acknowledge POs, provide ASN/invoice data.

Common user types within each organization - Planners: need forward-looking visibility—capacity, forecasts, purchase orders. - Operations: need day-to-day details—stops, ETAs, exceptions. - Business analysts: need trends and history—on-time performance, cost, root causes.

Data visibility examples - Private notes: an operations note on a shipment visible only inside the shipper's tenant. - Shared context: a delivery appointment visible to both the shipper and the carrier for a specific shipment. - Role-filtered: analysts see aggregated KPIs, planners see open exceptions, operators see actionable tasks.

Chapter 78. Identity and access: meet partners where they are

Every organization may authenticate differently: - Enterprise SSO (OIDC/SAML) for shippers and large suppliers. - Username/password for smaller partners. - Service accounts and tokens for system-to-system integrations.

Quantum supports these patterns and lets each tenant manage its own users (delegated administration). Permissions can differ by user type and tenant while staying auditable and predictable.

Chapter 79. Modeling without jargon: Areas, Domains, and Actions

To keep your platform organized, Quantum groups things into: - Functional Areas: broad business spaces like Collaboration, Finance, or Catalog. - Functional Domains: specific entity types within an area—Partner, Shipment, PurchaseOrder, Invoice. - Functional Actions: what users do—VIEW, CREATE, UPDATE, DELETE, APPROVE, EXPORT, etc.

Why this matters: - Clear menus for the UI (group screens by area and domain). - Clear policy rules (easier to say “Carriers can VIEW Shipments” or “Only Finance can APPROVE Invoices”).

Chapter 80. Policies that say “who can do what” (Rule Language)

Policies are simple, readable rules that match: - Who is calling (identity and roles) - What they’re trying to access (area, domain, action) - Request details (headers/body, like a shipment id or tenant)

Rules then allow or deny the action and can add filters so the user only sees data they’re permitted to see. See the Permissions guide for authoring details ([Permissions Guide](#)).

Write-time data placement (where a new record belongs) - By default, new records use the creator’s organization. - You can override per area/domain with a small policy—for example, keep Partner records in a shared “directory” domain while Shipments stay tenant-local. See “Data domain assignment on create” ([../user-guide/permissions.pdf](#)).

Chapter 81. Domain contexts and domain policies (placing and sharing data)

In supply chains, the same entity often lives in different business contexts (e.g., a Shipment inside a Supplier's operations vs. the Buyer's view). Domain contexts let you keep data where it naturally belongs while still collaborating safely:

- Keep most data tenant-local: Each organization owns its operational truth (orders, shipments, tasks).
- Create shared contexts deliberately: Use a shared "Directory" context for master data like Partners and Facilities so everyone references the same records.
- Policy-driven sharing: Collaboration bubbles share just the relevant records (a specific PO, shipment, or appointment) with the right partners. Policies ensure each party only sees the fields and actions appropriate to them.
- Regional and compliance contexts: Pin data to a geography (e.g., EU) when regulations require it; policies keep reads and writes aligned.

Business impact - Lower friction: Organize data the way teams actually work (local operations + shared directories). - Safer collaboration: Sharing is explicit and explainable—no surprise data leaks. - Compliance by construction: Residency and scoping rules are enforced by policy, not by custom code.

Chapter 82. Bootstrap tenants with Seed Packs (zero-touch baseline data)

Onboarding partners is a bottleneck in many SaaS rollouts. Seed packs provide a fast, repeatable way to initialize each tenant with the right starting data and settings.

What seed packs give you - Versioned, reviewable baseline: Curate CSV/JSON datasets and settings (code lists, roles, sample workflows) under version control. - Archetypes for editions: Bundle multiple packs into a named edition (e.g., “Logistics Core” vs. “Supplier Lite”) to match your product tiers. - Safe, idempotent updates: Re-apply packs any time; only changes are applied. You can inspect what’s pending before you apply. - Admin APIs and startup automation: New tenants can be provisioned with the right archetype automatically; admins can check pending updates and apply them when ready.

Why this matters to a supply-chain SaaS - Faster go-live: New buyers, suppliers, and carriers start with consistent roles, code lists, and sample workflows. - Less manual setup: Reduce back-and-forth with partners; eliminate error-prone spreadsheets and one-off scripts. - Controlled change: Roll out new catalogs or policy tweaks as a versioned pack; apply to selected tenants when they opt in.

Learn more: [Seed packs and declarative tenant seeding](#).

Chapter 83. Security policies: practical protections and proofs

Security isn't just login. The framework's policy layer turns business rules into enforceable, auditable protections across UI, APIs, and imports/exports.

What's protected - Least privilege by role: Planners, Operators, and Analysts only get the actions and fields they need. - Field-level and record-level controls: Share a shipment with a carrier without exposing private buyer notes or margins. - Tamper-evident changes: Important state transitions and task completions are tracked and explainable. - Delegated administration: Each organization manages its users and roles; platform admins retain oversight with audit trails.

Why it's better than ad-hoc controls - One policy engine, everywhere: The same rules guard screens, APIs, and bulk operations—reducing gaps and regressions. - Explainable decisions: When an action is denied or filtered, admins can see why. - Easier audits: Policies and seed packs create a repeatable, documented configuration per tenant.

Chapter 84. A small, powerful API surface:

List + Query

Instead of building a unique search endpoint for every screen, Quantum gives you: - List API: a single, consistent endpoint per domain to list, filter, sort, page, and project fields. - Query Language: a simple filter syntax so UIs and reports can ask precise questions. - Automatic enforcement: policies and data-domain rules are always applied server-side, so callers only receive allowed data.

Business outcomes - Faster delivery: new screens reuse the same list API. - Fewer mistakes: less custom code, more consistent results. - Safer by default: even power users can't bypass policy enforcement.

Chapter 85. Delegated Administration (tenant-level user management)

Empower each organization to run their own house while you keep platform-wide safety: - Tenant admins invite users, reset passwords, and assign roles. - Role templates per tenant align with their org structure (Planner, Ops, Analyst, Carrier Dispatcher, etc.). - Cross-tenant boundaries are respected; global administrators can still support, audit, and troubleshoot with impersonation/acting-on-behalf-of where permitted and logged.

Chapter 86. Integrations and data management

Supply chains depend on clean, timely data. Quantum provides:

- CSV imports/exports: onboard master data quickly, rerun safely, and let business users fix and re-upload.
- Stateful objects: model processes (e.g., Shipment lifecycle) with clear states—Created → In-Transit → Delivered → Closed.
- Completion Tasks: checklist-like steps (confirm pickup, upload POD, reconcile invoice) that drive work to done and provide accountability.
- Consistent access: the same policies that protect your UI protect imports/exports and API calls.

Example uses

- Bulk load a new supplier catalog with CSV import; analysts export exceptions weekly for review.
- Track shipment exceptions as tasks; operations completes them with evidence (attachments/notes), all audited.

Chapter 87. End-to-end examples

1) Buyer–supplier collaboration on a Purchase Order - Create a collaboration bubble around a PO so both parties see schedule, holds, and documents. - Supplier can UPDATE promised dates; buyer can APPROVE changes. Private buyer notes remain private.

2) Shared partner directory, curated centrally - Keep one shared Partner domain so everyone finds the same carrier and facility records. - Only directory curators can CREATE/UPDATE; all tenants can VIEW.

3) EU shipment residency - Shipments created by anyone in Europe are written to an EU partition by policy. Reads remain role- and tenant-scoped.

Chapter 88. What you don't have to build from scratch

- Data isolation and safe sharing across tenants
- A consistent CRUD and search API for every domain
- A policy engine that explains its decisions and applies filters
- A write-time placement policy (so data lands in the right partition)
- Patterns for long-running, stateful business processes and task completion

The framework gives you these foundations so your teams focus on business value—on-time deliveries, lower cost, happier customers.

Chapter 89. Other helpful capabilities you get out of the box

- Pending seed visibility: Admins can see which seed packs have updates pending for a tenant before applying them, reducing surprise changes.
- Version selection and rollback strategy: The platform automatically selects the latest compatible seed pack version; because packs are versioned and reviewable, you can test in a sandbox realm before rolling out.
- Archetypes (editions): Model product tiers by composing packs; useful for offering lighter supplier/carrier editions versus full buyer editions.
- Explainable access decisions: When access is denied or results are filtered, admins can inspect the reasoning—useful during partner onboarding and audits.
- Idempotent imports/exports: Bulk data operations can be retried safely, which is critical when partners exchange large files over unreliable networks.
- Observability hooks: Standardized endpoints and logs make it easier to monitor SLAs (e.g., late tasks, delayed updates) without custom plumbing.

Chapter 90. Next steps

- Start with siloed defaults; prove value quickly using the List API.
- Add small, targeted policies to enable collaboration bubbles and shared directories.
- Introduce delegated administration so partners self-serve.
- Use CSV imports and Completion Tasks to operationalize data stewardship.
- Deep dive: Permissions and Rule Language ([Permissions Guide](#)), and Data domain assignment on create ([Data domain assignment on create](#)).

Chapter 91. A day in the life: From Purchase Order to Delivery

This story ties the pieces together in a realistic sequence. We follow a Purchase Order (PO) from creation to delivery, with shared visibility for suppliers and carriers, a clear state graph, and checklist-like Completion Tasks guiding the work.

1) Purchase Order is created (by the Buyer) - Action: A buyer creates a PO in the Collaboration area under the PurchaseOrder domain. - Data placement: By default, the PO is written to the buyer's organization (their data domain). If you prefer a shared domain for POs, configure a small policy; otherwise, the default works well. - Programmatic sharing: A rule shares the specific PO with the chosen Supplier (or Suppliers). The Supplier can view the PO and update the fields you allow (e.g., promised date), but cannot see private buyer-only fields.

State graph (illustrative) - Draft → Open → SupplierAcknowledged → ReadyToShip → PartiallyShipped → FullyShipped → Received → Closed

Completion Tasks (examples attached to the PO) - Buyer: Provide required documents (commercial terms, incoterms) - Supplier: Acknowledge PO (due in 24 hours) - Supplier: Provide ASN (advanced shipping notice) for each shipment - Supplier: Confirm pickup window - Buyer: Approve any date changes

2) The Supplier prepares shipments (shared onward to Carriers) - Action: The Supplier creates one or more Shipments linked to the PO (and optionally to specific lines). - Data placement: Shipments are written to the Supplier's domain by default (their own organization), but are shared with the Buyer so both parties see the same timeline. - Sharing to Carriers: When the Supplier tenders a shipment, the shipment is shared with the selected Carrier so they can update movement and milestones.

Shipment state graph (illustrative) - Planned → Tendered → Accepted → InTransit → Delivered → ProofVerified → Closed

Shipment Completion Tasks (examples) - Carrier: Confirm pickup - Carrier: Update in-transit location/ETA - Carrier: Upload POD (proof of delivery) - Supplier: Reconcile quantities shipped vs. ordered

3) Status updates complete tasks and move states forward - When the Supplier marks "SupplierAcknowledged," the PO's acknowledgement task completes and the PO moves to SupplierAcknowledged. - When all lines are ready and at least one shipment is created, the PO advances to ReadyToShip. If some but not all lines ship, it enters PartiallyShipped; once all lines ship, it becomes FullyShipped. - Carrier updates (e.g., Delivered with POD uploaded) complete shipment tasks. Those completion events can also advance the PO state (e.g., all shipments Delivered → PO moves to Received). Final checks (invoices matched, discrepancies resolved) move the PO to Closed.

Why this is safe and predictable - Roles and policies ensure each party sees only what they should: the Buyer sees everything; the Supplier sees the shared PO and its related shipments; the Carrier sees only the shipments they handle. - Completion Tasks remove ambiguity: everyone knows the

next step and who owns it. Each task completion is audited. - The state graph makes lifecycle transitions explicit. Policies can require certain tasks to be completed before a state transition is allowed.

4) Business visibility and reporting (List API + Query) - Operations view: “Purchase Orders in progress” shows all POs in Open, SupplierAcknowledged, ReadyToShip, or PartiallyShipped, including late tasks and upcoming milestones. - Buyer/supplier view: Both parties see the same PO status and related Shipments, with role-appropriate fields. - Simple reporting example (illustrative):
GET /collaboration/purchaseorder/list?filter=status IN
("Open","SupplierAcknowledged","ReadyToShip","PartiallyShipped")&sort=dueDate:asc&limit=50 -
Add projections to include key fields and roll-ups (e.g., shipped vs. ordered quantities). Related Shipment info can be retrieved similarly via the List API on the Shipment domain, filtered by the PO id.

What made this easy (and repeatable) - Multi-tenancy by default: Each org’s data is isolated; sharing is explicit and safe. - Policies (Rule Language): Define who can see or update which fields and when. The same rules apply to UI, API, and imports/exports. - Data domain assignment on create: Defaults keep data in the creator’s org; you can configure exceptions (e.g., shared directories) with a tiny policy. - Stateful objects + Completion Tasks: Clear states and checklists turn complex collaboration into a predictable flow. - List API + Query Language: One consistent way to fetch work lists, timelines, and reports without proliferating custom endpoints.

Chapter 92. Appendix

92.1. Configuration Reference

This appendix explains key configuration properties used in Quantum applications, organized by functional area.

92.1.1. Configuration Basics

File Locations

- `src/main/resources/application.properties` - main configuration
- `src/test/resources/application.properties` - test overrides
- `.env` file - local development secrets (not committed)

Profiles and Environment Variables

```
# Profile-specific properties
%dev.auth.jwt.duration=7200
%test.quarkus.mongodb.database=test-db
%prod.quarkus.log.level=WARN

# Environment variable substitution
quarkus.mongodb.connection-
string=${MONGODB_CONNECTION_STRING:mongodb://localhost:27017}
auth.jwt.secret=${JWT_SECRET:change-me-in-production}
```

Precedence

System properties > Environment variables > application.properties

92.1.2. Database Configuration

MongoDB Connection

```
# Basic connection
quarkus.mongodb.connection-string=mongodb://localhost:27017
quarkus.mongodb.database=my-app

# With authentication
quarkus.mongodb.connection-
string=mongodb://user:pass@host:27017/database?authSource=admin

# Disable dev services (use real MongoDB)
quarkus.mongodb.devservices.enabled=false
```

Morphia Configuration

```
# Database and package scanning
quarkus.morphia.database=my-app-db
quarkus.morphia.packages=com.example.model,com.example.entities

# Schema management
quarkus.morphia.create-caps=true
quarkus.morphia.create-indexes=true
quarkus.morphia.create-validators=true
```

92.1.3. Authentication Configuration

JWT Provider

```
# Enable JWT authentication
quarkus.smallrye-jwt.enabled=true
auth.provider=custom

# JWT validation
mp.jwt.verify.publickey.location=publicKey.pem
mp.jwt.verify.issuer=my-app
mp.jwt.verify.audiences=my-app-users

# Custom JWT settings
auth.jwt.secret=${JWT_SECRET:your-secret-here}
auth.jwt.expiration=60
auth.jwt.refresh-expiration=1440
```

OIDC Provider

```
# Enable OIDC
quarkus.oidc.enabled=true
auth.provider=oidc

# OIDC configuration
quarkus.oidc.auth-server-url=https://your-provider.com/auth/realms/your-realm
quarkus.oidc.client-id=your-client-id
quarkus.oidc.credentials.secret=your-client-secret
quarkus.oidc.roles.role-claim-path=groups
```

92.1.4. Quantum Framework Configuration

Database Migration

```
# Migration settings
```

```
quantum.database.version=1.0.0
quantum.database.scope=DEV
quantum.database.migration.enabled=true
quantum.database.migration.changeset.package=com.example.migrations
```

Realm and Tenant Configuration

```
# System realm configuration
quantum.realmConfig.systemTenantId=system
quantum.realmConfig.systemAccountId=system-account
quantum.realmConfig.systemOrgRefName=SYSTEM
quantum.realmConfig.systemUserId=system-user

# Default tenant settings
quantum.realmConfig.defaultTenantId=default
quantum.realmConfig.defaultAccountId=default-account
quantum.realmConfig.defaultOrgRefName=DEFAULT

# Anonymous user (use carefully)
quantum.anonymousUserId=anonymous
```

92.1.5. See Also

- [Quarkus Configuration Guide](#)
- [Authentication Configuration](#)

92.1.6. Ontology Configuration (optional)

Enablement

```
e2eq.ontology.enabled=false # default; set true to enable ontology wiring in your app
```

Registry modes

- In-memory (recommended to start): build a TBox in code/resources and instantiate `OntologyRegistry.inMemory(tbox)`.
- Mongo-backed (advanced): load a TBox from a Mongo collection and cache it in memory for hot reload/versioning.

Edges collection and indexes

- Create an edges collection (name of your choice) when enabling ontology. Ensure the following indexes:
- Compound: { tenantId:1, p:1, dst:1 }
- Compound: { tenantId:1, src:1, p:1 }

- Optionally, add a TTL on ts for inferred edges if you plan periodic full recomputes.

Materialization hooks

- Use `OntologyMaterializer` to recompute inferred edges when entities change (sources and intermediates for your property chains).

See also

- [Ontologies in Quantum](#)
- [Integrating Ontology](#) == Quarkus Foundation

Quantum builds on Quarkus, a Kubernetes-native Java stack optimized for fast startup, low memory footprint, and developer productivity.

92.1.7. Key Quarkus Capabilities

Developer Experience

- **Live reload:** Code changes are reflected immediately in dev mode
- **Unified config:** Single `application.properties` for all configuration
- **Dev UI:** Rich development interface at <http://localhost:8080/q/dev/>
- **Test-first ergonomics:** Built-in test support with `@QuarkusTest`

Build-time Optimizations

- **Classpath indexing:** Aggressive build-time analysis reduces runtime scanning
- **Ahead-of-time processing:** Framework initialization moved to build time
- **Jandex indexes:** Annotation discovery without runtime reflection

Cloud Native

- **Container integration:** Seamless Docker and Kubernetes deployment
- **Health checks:** Built-in health and readiness endpoints
- **Metrics:** Micrometer and Prometheus integration
- **Configuration:** Environment-aware config with profiles

92.1.8. Arc (Quarkus CDI)

Arc is Quarkus' CDI implementation, focused on build-time analysis and small runtime overhead.

Common Scopes

Scope	Usage
<code>@ApplicationScoped</code>	Stateless services, repositories (recommended default)
<code>@RequestScoped</code>	Per-request context holders

Scope	Usage
@Singleton	Single instance without CDI proxies
@Dependent	Short-lived helpers (default if no scope declared)

Bean Discovery

- Classes become beans with scope annotations or producer methods
- Use @Qualifier to disambiguate multiple implementations
- Instance<T> for programmatic bean selection

Example

```
@ApplicationScoped
public class ProductService {
    @Inject ProductRepo repo;

    public List<Product> findActive() {
        return repo.findByActive(true);
    }
}
```

92.1.9. Native Compilation with GraalVM

Benefits

- Millisecond startup times
- Drastically reduced memory footprint
- Ideal for serverless and microservices

Constraints

- Reflection needs explicit configuration
- Dynamic proxies require substitution
- Some dynamic classloading limitations

Configuration

Use @RegisterForReflection for classes accessed via reflection:

```
@RegisterForReflection
@Entity
public class Product extends BaseModel {
    // Ensures reflection metadata in native image
}
```

92.1.10. Extension Ecosystem

Quarkus provides extensions for: - **Data**: MongoDB, PostgreSQL, Redis, Elasticsearch - **Security**: JWT, OIDC, OAuth2, RBAC - **Messaging**: Kafka, AMQP, JMS - **Observability**: Metrics, tracing, health checks - **Cloud**: AWS, Azure, Google Cloud integrations

92.1.11. Quantum's Quarkus Integration

Quantum leverages Quarkus for: - **CDI**: Dependency injection for repositories and services - **JAX-RS**: REST endpoint framework via BaseResource - **Security**: JWT/OIDC integration for authentication - **MongoDB**: Morphia integration for persistence - **Configuration**: MicroProfile Config for settings - **OpenAPI**: Automatic API documentation

92.1.12. Getting Started with Quarkus

Create Project

```
mvn io.quarkus:quarkus-maven-plugin:create \
  -DprojectId=com.example \
  -DprojectArtifactId=my-app \
  -DclassName="com.example.MyResource" \
  -Dpath="/hello"
```

Development Mode

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

Native Build

```
./mvnw package -Pnative
```

92.1.13. See Also

- [Quarkus Guides](#)
- [CDI Reference](#)
- [Native Image Guide](#) == Glossary

92.1.14. Core Concepts

BaseModel

Abstract base class for all Quantum entities, providing DataDomain, audit fields, ID management, and validation hooks.

DataDomain

Scoping information attached to every model (tenantId, orgRefName, ownerId, etc.) that enables multi-tenant isolation and controlled sharing.

DomainContext

Runtime context representing the current execution scope (tenant, org, user, functional area/domain, action) used by repositories and security rules.

Functional Area

Broad business capability grouping (e.g., Catalog, Collaboration, Identity) used for organizing models and security policies.

Functional Domain

Specific entity type within a functional area (e.g., Product within Catalog, Shipment within Collaboration) used for fine-grained permissions.

RuleContext

Policy evaluation engine that determines allowed actions and contributes data filters based on identity, functional area/domain, and business rules.

92.1.15. Multi-Tenancy

Tenant

Logical partition representing a customer or organization in a multi-tenant system, identified by tenantId.

Organization (Org)

Business unit within a tenant, identified by orgRefName, enabling sub-tenant grouping and sharing.

Realm

Database or data partition, often corresponding to a MongoDB database, that can be selected per request via X-Realm header.

Shared Domain

Data domain accessible across multiple tenants (e.g., public catalogs, partner directories) as opposed to tenant-isolated data.

92.1.16. Security

Principal

Authenticated user identity with associated roles, tenant membership, and permissions.

Permission Rule

Declarative policy that matches requests (URL, method, headers, body) and decides ALLOW/DENY with optional data filters.

Access Resolver

Plugin that computes dynamic access lists (e.g., customer IDs a user can see) for use in permission filters.

Impersonation

Acting as another user identity for troubleshooting or administrative purposes, controlled by scripts and realm restrictions.

92.1.17. Data and Persistence

MorphiaRepo

Repository interface extending MongoDB operations with Quantum's DataDomain filtering, validation, and audit capabilities.

EntityReference

Lightweight reference object containing ID, type, refName, and displayName, used for foreign keys without full object loading.

StateGraph

Finite state machine definition for model fields, enforcing valid states and transitions (e.g., Order: Draft → Processing → Shipped).

Completion Task

Persistent work item with status tracking, used for checklists and long-running processes with audit trails.

92.1.18. REST and APIs

BaseResource

Abstract REST resource class providing consistent CRUD endpoints (find, get, list, save, update, delete) with automatic security and validation.

Query Language

ANTLR-based filter syntax used across all list endpoints, permission rules, and access resolvers for consistent data querying.

UIActions

List of actions a user can perform on a specific entity instance, computed based on entity state and user permissions.

CSV Import/Export

Built-in endpoints for bulk data operations with validation, preview sessions, and error handling.

92.1.19. Framework Components

ValidationInterceptor

Morphia interceptor that runs Jakarta Bean Validation and defaults DataDomain before persistence.

SecurityFilter

JAX-RS filter that builds security context (PrincipalContext, ResourceContext) and evaluates permissions for each request.

DataDomainResolver

Service that determines which DataDomain to assign to new entities based on functional area/domain policies.

Migration

Versioned database schema and data changes managed by Quantum's migration framework with changeset tracking.

92.1.20. Annotations

@FunctionalMapping

Class-level annotation declaring a model's functional area and domain, replacing legacy bmFunctionalArea() methods.

@FunctionalAction

Method-level annotation specifying the action performed by a REST endpoint when it differs from HTTP verb defaults.

@TrackReferences

Field annotation on @Reference fields that maintains back-reference sets for referential integrity checking.

@RegisterForReflection

Quarkus annotation ensuring classes are available for reflection in native images.

92.1.21. External Integrations

JWT Provider

Authentication module that validates JSON Web Tokens and populates security context with user identity and roles.

OIDC Integration

OpenID Connect support for enterprise identity providers like Keycloak, Auth0, and AWS Cognito.

Feature Flags

Configuration-driven capability toggles with targeting rules for gradual rollouts and A/B testing.

Postmark Integration

Email service integration for transactional messaging with template support.

92.1.22. Ontology (optional feature)

Ontology

A formal description of concepts (classes) and their relationships (properties/predicates) with rules (e.g., property chains) that allow inferring implied facts. See [Ontologies in Quantum](#).

Predicate / Property

A named relationship (e.g., placedBy, memberOf, orderShipsToRegion). Predicates connect source and destination IDs and can be declared inverse or transitive.

Property Chain

A rule of the form $p \sqcap q \Rightarrow r$ meaning if $(A \dashv p \dashv B)$ and $(B \dashv q \dashv C)$, infer $(A \dashv r \dashv C)$. Chains make common traversals first-class and fast.

Edge (materialized)

A persisted tuple (tenantId, src, p, dst) representing a relationship, often inferred from rules. Edges live in a dedicated collection and power single-hop queries and policies.

Ontology Materializer

A component that computes inferred edges for an entity snapshot and upserts them to the edges collection. Keeps edges fresh as data changes.

ListQueryRewriter

A helper that turns semantic constraints into efficient Mongo queries (e.g., id IN srcIdsByDst). Integrates with permission rules and Morphia repos.