Table of Contents

Chapter 1. Modeling with Functional Areas, Domains, and Actions

Quantum organizes your system around three core constructs:

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

These constructs allow:

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

1.1. Prefer Annotations for Functional Mapping (recommended)

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

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

Example: model annotated with FunctionalMapping

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

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

```
@FunctionalMapping
}
```

Example: resource method annotated with FunctionalAction

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

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

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

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

How the framework uses these annotations

- SecurityFilter: If the matched resource class has @FunctionalMapping, it uses area/domain from the annotation. If the method has @FunctionalAction, it uses that value; otherwise, it infers the action from the HTTP method (GET=VIEW, POST=CREATE, PUT/PATCH=UPDATE, DELETE=DELETE). If annotations are absent, it falls back to the existing path- and convention-based logic.
- MorphiaRepo.fillUIActions: If the model class has @FunctionalMapping, its area/domain are used to resolve allowed UI actions; otherwise, it falls back to the legacy bmFunctionalArea()/bmFunctionalDomain() methods.
- PermissionResource: When listing functional domains, it prefers @FunctionalMapping on entity classes and falls back to bmFunctionalArea()/bmFunctionalDomain() when missing.

Migration notes

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

See also: Security Annotations: FunctionalMapping and FunctionalAction

1.2. DataDomain on Models

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

Example model:

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

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

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

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

1.3. Persistence Repositories

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

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

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

1.4. Exposing REST Resources

Expose consistent CRUD endpoints by extending BaseResource.

```
import com.e2eq.framework.rest.resources.BaseResource;
```

```
import jakarta.ws.rs.Path;

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

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

1.5. Lombok in Models

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

Common annotations you will see:

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

Example:

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

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

1.6. Validation with Jakarta Bean Validation

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

Typical annotations:

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

Where validation runs:

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

1.7. Jackson vs Jakarta Validation Annotations

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

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

Correspondence and interplay:

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

Chapter 2. 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);
  }
```

```
}
```

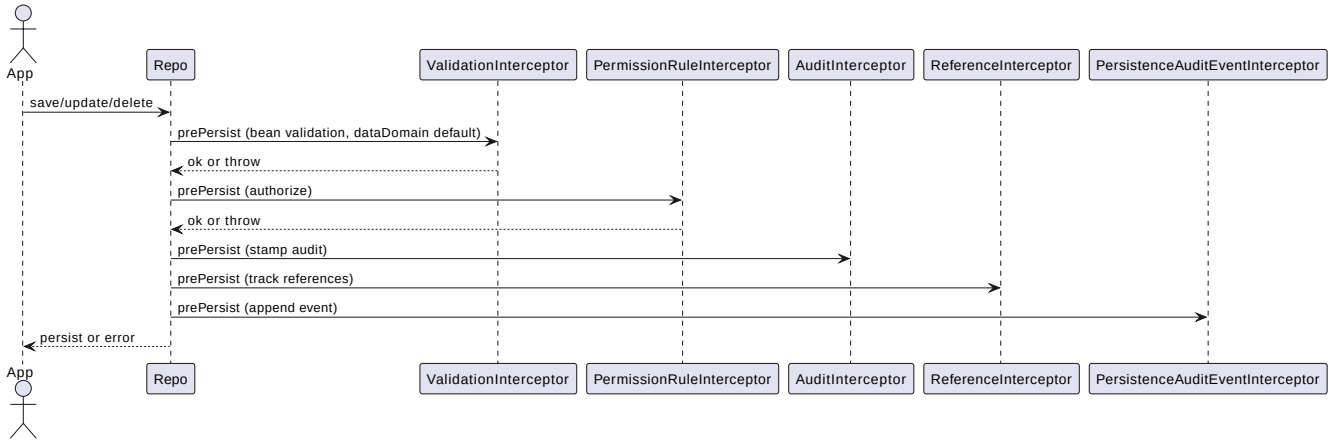
- Or set properties in application.properties:

```
# 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 3. 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

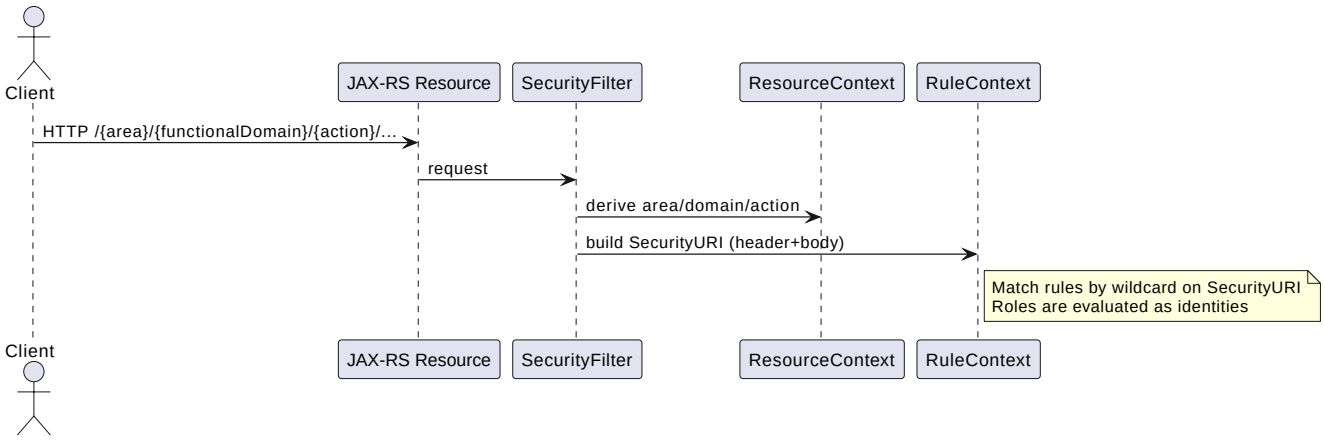
High-level responsibilities:

- 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 4. 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 5. StateGraphs on Models

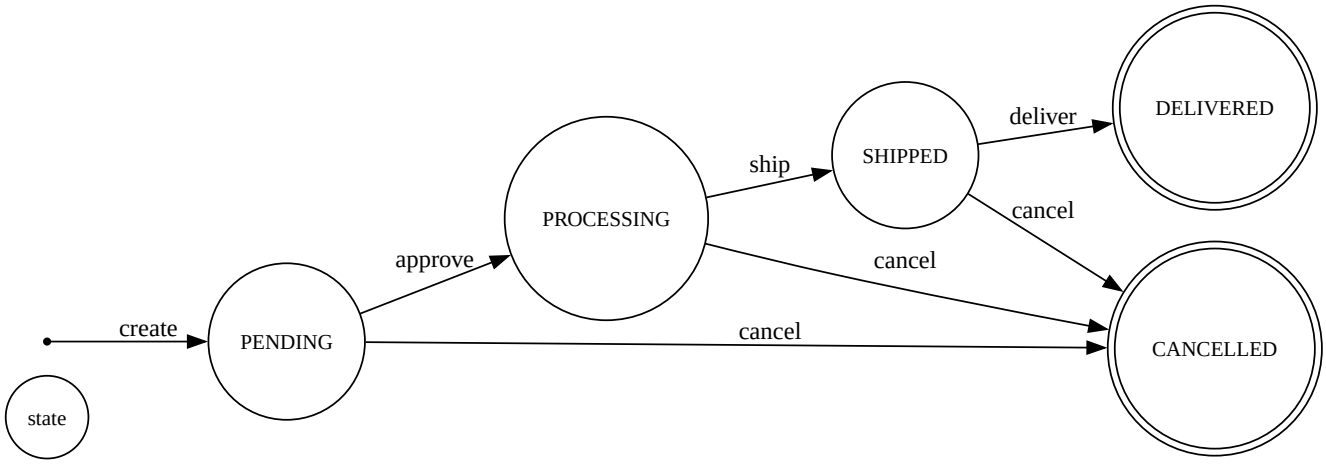
StateGraphs let you restrict valid values and transitions of String state fields. They are declared on model fields with @StateGraph and enforced during save/update when the model class is annotated with @Stateful.

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:

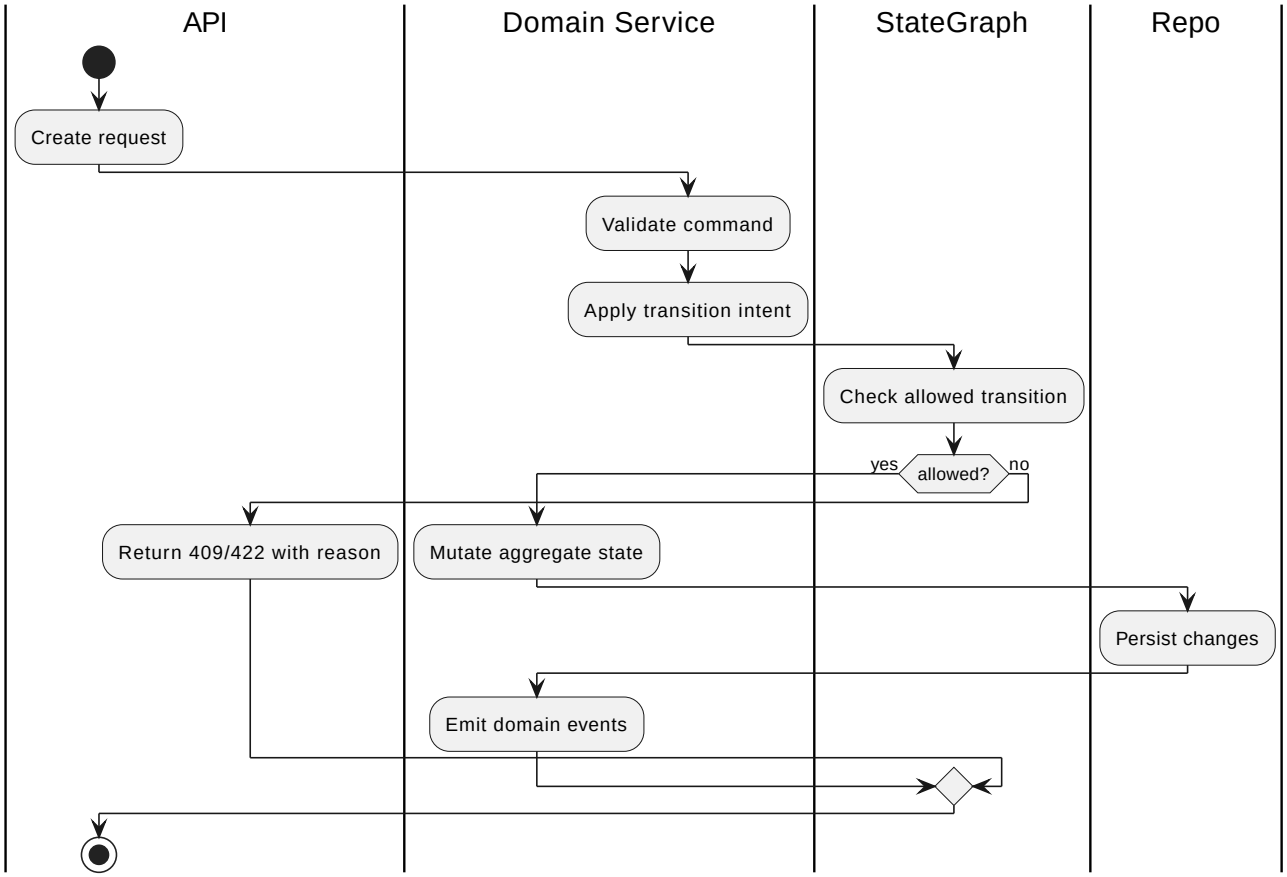
```
@Startup
@ApplicationScoped
public class StateGraphInitializer {
  @Inject StateGraphManager stateGraphManager;
  @PostConstruct void init() {
    StringState order = new StringState();
    order.setFieldName("orderStringState");

    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 6. 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 7. 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.

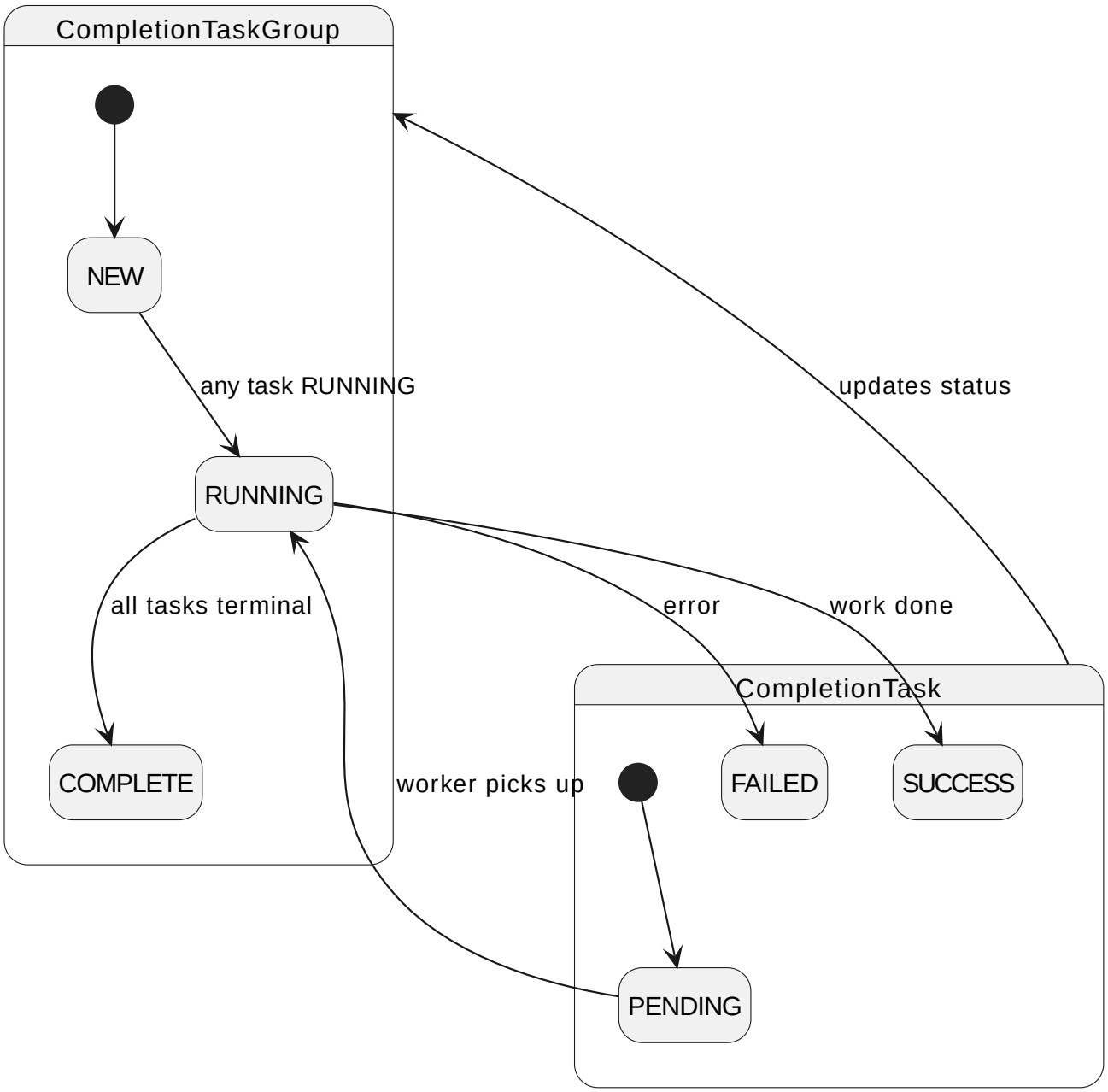
Key models:

- 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 createdDate;            // 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 createdDate;     // 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 8. 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 9. 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.

9.1. 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).

9.1.1. 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 --p-→ B) and (B --q-→ C) then we infer (A --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.

9.1.2. 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.

9.1.3. 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 --p-→ B & B --q-→ C ⇒ A --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.

9.1.4. 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 → 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).

9.1.5. 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 → B → C, define a chain p□q ⇒ r and materialize r for faster queries and simpler policies.
- Capture inverses and transitivity
- For natural inverses (parentOf □ 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.

9.1.6. 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 Order → User → 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 □ memberOf ⇒ 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 □ ancestorOf ⇒ placedInOrg to include parents automatically.

9.1.7. 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.

9.1.8. 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.

9.1.9. 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.

9.1.10. 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.

9.1.11. 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 □ memberOf ⇒ placedInOrg
- If (order --placedBy-→ customer) and (customer --memberOf-→ org), then infer (order --placedInOrg-→ org)
- orderHasShipment □ shipsTo ⇒ orderShipsTo
- If (order --orderHasShipment-→ shipment) and (shipment --shipsTo-→ address), infer (order --orderShipsTo-→ address)
- orderShipsTo □ locatedIn ⇒ orderShipsToRegion
- If (order --orderShipsTo-→ address) and (address --locatedIn-→ region), infer (order --orderShipsToRegion-→ region)
- placedInOrg □ ancestorOf ⇒ 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 ⇒ derived edge from task to order; combine with taskProduces to connect order to shipment without touching Order fields:
- fulfills taskProduces ⇒ orderHasShipment
- realizedBy orderHasShipment ⇒ fulfilledByUnit
- If (order --realizedBy-→ fu) and (order --orderHasShipment-→ s) ⇒ (fu --fulfillsShipment-→ s) or simply (order --fulfilledByUnit-→ 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 ⇒ openReturnOnOrg via placedInOrg:
- hasReturn placedInOrg ⇒ returnPlacedInOrg
- returnFor orderShipsToRegion ⇒ 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.

9.2. 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.

9.2.1. 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.

9.2.2. 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.

9.2.3. 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)

9.2.4. 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);
  }
}
```

9.2.5. 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.

9.2.6. 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.

9.2.7. 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.

9.2.8. 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.

9.2.9. 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.