

# Modeling with Functional Areas, Domains, and Actions

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

# Table of Contents

1. DataDomain on Models .....	2
2. Persistence Repositories .....	3
3. Exposing REST Resources .....	4
4. Lombok in Models .....	5
5. Validation with Jakarta Bean Validation .....	6
6. Jackson vs Jakarta Validation Annotations .....	7
7. Jackson ObjectMapper in Quarkus and in Quantum .....	8
8. Validation Lifecycle and Morphia Interceptors .....	10
9. Functional Area/Domain in RuleContext Permission Language .....	11
10. StateGraphs on Models .....	13
11. References and EntityReference .....	15
12. Tracking References with @TrackReferences and Delete Semantics .....	16

Quantum organizes your system around three core constructs:

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

These constructs allow:

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

# Chapter 1. DataDomain on Models

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

Example model:

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

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

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

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

# Chapter 2. Persistence Repositories

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

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

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

# Chapter 3. Exposing REST Resources

Expose consistent CRUD endpoints by extending `BaseResource`.

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

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

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

# Chapter 4. Lombok in Models

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

Common annotations you will see:

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

Example:

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

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

# Chapter 5. Validation with Jakarta Bean Validation

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

Typical annotations:

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

Where validation runs:

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



# Chapter 6. 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 7. 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 8. 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 9. Functional Area/Domain in RuleContext Permission Language

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

How these map into authorization and rules:

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

Examples

## 1) Header-based rule matching (Permissions)

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

## 2) Query variable usage (Filters)

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

branching in custom rule evaluators):

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

### 3) Model-driven mapping

Given a model like:

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

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

#### Notes

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

# Chapter 10. StateGraphs on Models

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

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

Defining a state graph at startup:

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

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

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

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

Using the graph in a model:

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

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

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



# Chapter 11. 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 12. 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.