Table of Contents

Chapter 1. Getting Started: Your First Quantum Application

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

1.1. Prerequisites

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

1.2. Project Setup

Create a new Quarkus project with Quantum dependencies:

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

Add Quantum dependencies to `pom.xml`:

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

1.3. Configuration

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

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

Basic `application.properties`:

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

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

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

1.4. Your First Model

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

```
package com.example.model;

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

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

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

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

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

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

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

1.5. Repository

Create a repository interface:

```
package com.example.repository;

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

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

1.6. REST Resource

Create a REST endpoint:

```
package com.example.resource;

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

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

1.7. Running the Application

Start your application:

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

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

1.8. Testing Your API

Create a product:

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

List products:

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

Filter products:

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

1.9. 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, createdDate, lastUpdatedBy, lastUpdatedDate fields

Consistent APIs: Standard REST patterns across all resources

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

OpenAPI: Automatic API documentation

1.10. Next Steps

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

1.11. 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.