Table of Contents

Implementing Queries and Projections (10 pages)

> The best view comes after the hardest climb.
>
> — Anonymous

In the section on CQRS, we described how DDD and CQRS complement each other and how the query side (read models) can be used to create one or more representations of the underlying data. In this chapter, we will dive deeper into how we can construct read optimized representations of the data by listening to domain events. We will also look at persistence options for these read models.

When working with query models, we construct models by listening to events as they happen. We will examine how to deal with situations where:

- New requirements evolve over a period of time requiring us to build new query models.
- We discover a bug in our query model which requires us to recreate the model from scratch.

Technical requirements

- JDK 1.8+ (We have used Java 16 to compile sample sources)
- JavaFX SDK 16 and SceneBuilder
- Spring Boot 2.4.x
- mvvmFX 1.8 (https://sialcasa.github.io/mvvmFX/)
- JUnit 5.7.x (Included with spring boot)
- TestFX (for UI testing)
- OpenJFX Monocle (for headless UI testing)
- Project Lombok (To reduce verbosity)
- Axon server to act as an event store
- Maven 3.x

Query models

In chapter 5, we saw how CQRS applications make use of distinct models to service commands and queries respectively. We also saw how to publish events when a command is successfully processed. Now, let’s look at how we can construct a query model by listening to these domain events. Logically, this will look something like how it is depicted here:

[cqrs application] | *cqrs/cqrs-application.png*

Figure 1. CQRS application

On the command side:

1. A request to mutate state (command) is received.
2. In an event-sourced system, the command model is constructed by replaying existing events that have occurred for that instance. In a state-stored system, we would simply restore state by reading state from the persistence store.
3. If business invariants (validations) are satisfied, one or more domain events are published.
4. In an event-sourced system, the domain event is persisted on the command side. In a state-stored system, we would update the state of the instance in the persistence store.
5. The external world is notified by publishing these domain events.

On the query side:

1. An event listening component listens to these domain events.
2. Constructs a purpose-built query model to satisfy a specific query use case.
3. This query model is persisted in a datastore optimized for read operations.
4. This query model is then exposed in the form of an API.

Consuming events

Persisting a query (read) model

Creating additional query (read) models

Historic event replays