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

Continuing our design journey

In chapter 4 - Domain analysis and modeling, we discussed eventstorming as a lightweight method to clarify business flows. As a reminder, this is the output produced from our eventstorming session:

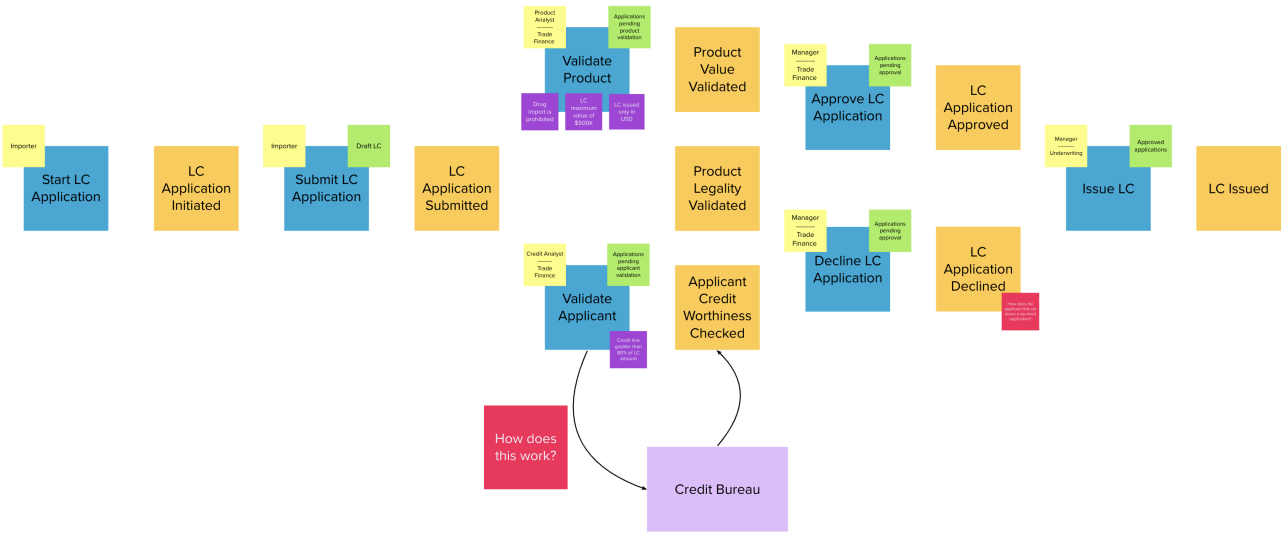

Figure 1. Recap of eventstorming session

As mentioned previously, the **green** stickies

Identifying queries

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:

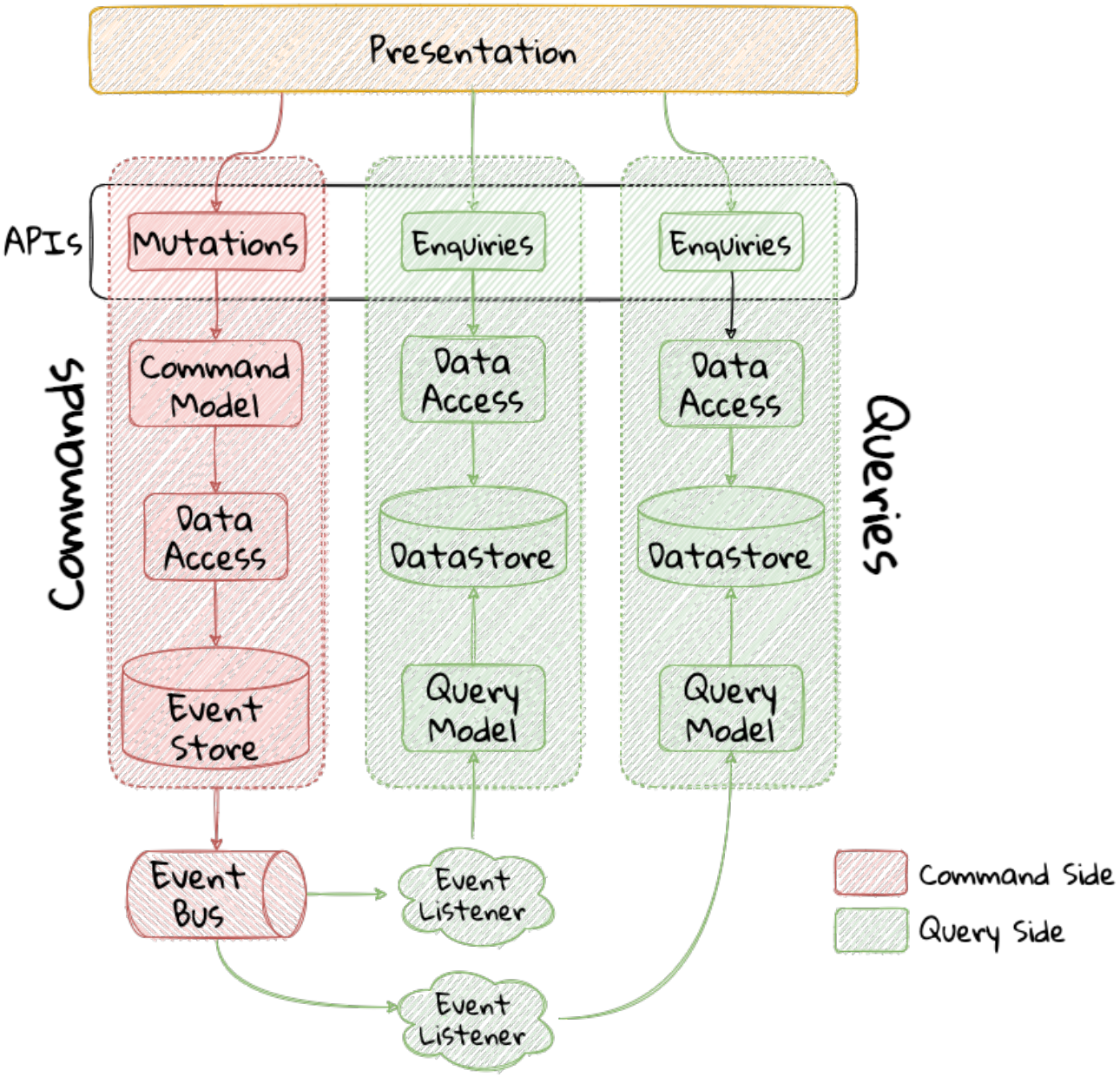

Figure 2. CQRS application

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 onto an event bus. The event bus is an infrastructure component onto which events are published.

We have already seen the command side flow in action in Chapter 5 - Implementing Domain Logic.

On the query side:

1. An event listening component listens to these domain events published on the event bus.
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.

Let's implement each of these steps to see how this works for our LC issuance application.

Consume events

As seen in chapter 5, when starting a new LC application, the importer sends a `StartNewLCApplicationCommand`, which results in the `LCApplicationStartedEvent` being emitted as shown here:

```
class LCApplication {
    //..
    @CommandHandler
    public LCApplication(StartNewLCApplicationCommand command) {
        // Validation code omitted for brevity
        // Refer to chapter 5 for details.
        AggregateLifecycle.apply(new LCApplicationStartedEvent(command.getId(),
                command.getApplicantId(), command.getClientReference()));
    }
    //..
}
```

Let's write an event processing component which will listen to this event and construct a query model. When working with the Axon framework, we have a convenient way to do this by annotating the event listening method with the `@EventHandler` annotation.

```
import org.axonframework.eventhandling.EventHandler;

class LCApplicationStartedEventHandler {

    @EventHandler                                          ①
    public void on(LCApplicationStartedEvent event) {      ②
        // Construct a query model here
    }
}
```

① To make any method an event listener, we annotate it with the `@EventHandler` annotation.

② The handler method needs to specify the event that we intend to listen to. There are other arguments that are supported for event handlers. Please refer to the Axon framework documentation for more information.

The `@EventHandler` annotation should not be confused with the `@EventSourcingHandler` annotation which we looked at in chapter 5. The `@EventSourcingHandler` annotation is used to replay events and restore aggregate state when loading event-sourced aggregates on the command side, whereas the `@EventHandler` annotation is used to listen to events outside the context of the aggregate. In other words, the `@EventSourcingHandler` annotation is used exclusively within aggregates, whereas the `@EventHandler` annotation can be used anywhere there is a need to consume domain events. In this case, we are using it to construct a query model.

Persisting a query (read) model

Creating additional query (read) models

Historic event replays

The need for replays

Types of replays

Full event replays

Partial event replays

Adhoc event replays

Event replay considerations

Application availability

Optimization techniques