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:

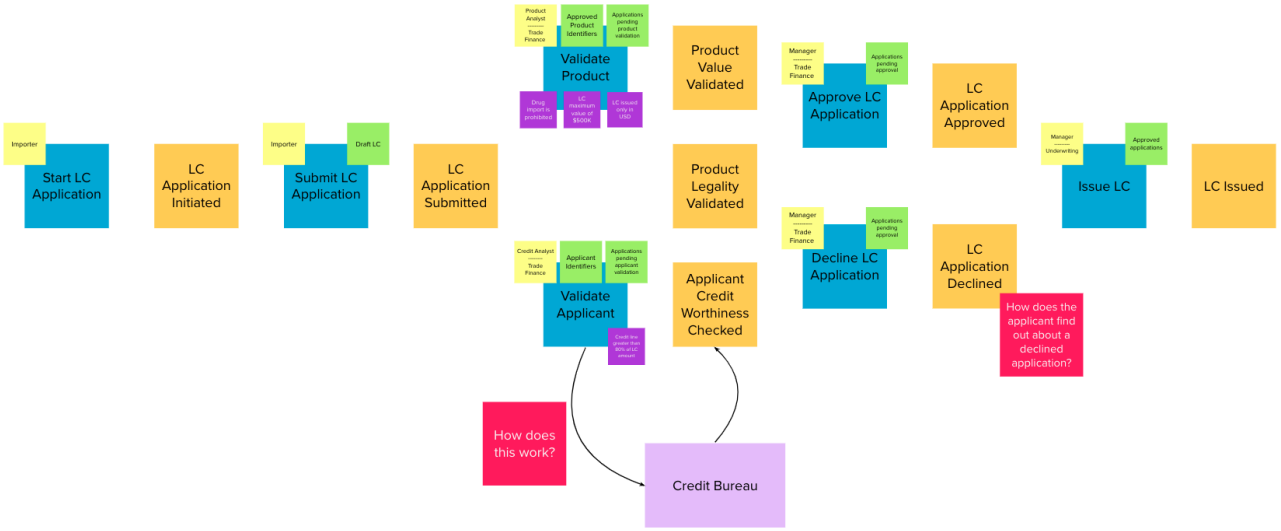

Figure 1. Recap of eventstorming session

As mentioned previously, we are making use of the CQRS architecture pattern to create the solution. For a detailed explanation on why this is a sound method to employ, please refer to the Why CQRS section in Chapter 5 - Implementing Domain Logic. In the diagram above, the **green** stickies represent **read/query models**. These query models are required when validating a command (for example: list of valid product identifiers when processing the `ValidateProduct` command) or if information is simply required to be presented to the user (for example: a list of LCs created by an applicant).

Applying CQRS

As covered in chapter 3, the CQRS pattern separates write (operations that mutate state) and read (operations that answer questions) operations into distinct (logical and/or physical) components from an architecture perspective. Let's look at what it means to apply CQRS in practical terms.

Why CQRS?

As we have seen previously, in a CQRS-based system, there are distinct sets of models:

- One for the command side
- One or more for the query side

In chapter 5, we 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:

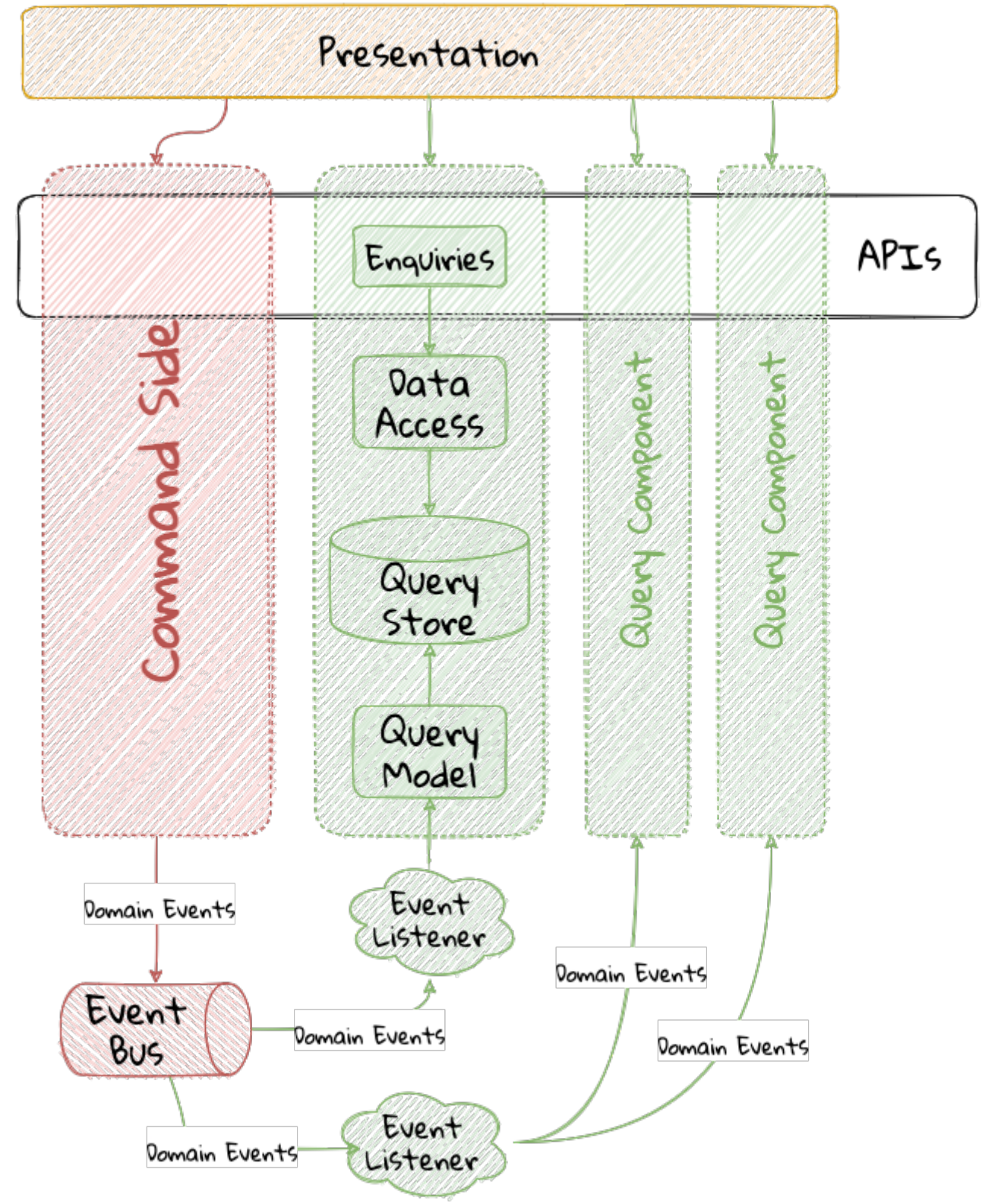

Figure 2. CQRS application — query side

Please refer to the section on implementing the command side in Chapter 5 for a detailed explanation of how the command side is implemented.

The high level sequence on the query side is described here:

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.

Note how there can exist more than one query side component to handle respective scenarios.

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

Tooling choices

In a CQRS application, there is a separation between the command and query side. At this time, this separation is logical in our application because both the command and query side are running as components within the same application process. To illustrate the concepts, we will use conveniences provided by the Axon framework to implement the query side in this chapter. In Chapter 10, we will look at how it may not be necessary to use a specialized framework (like Axon) to implement the query side.

Implementing the query side

When implementing the query side, we have two concerns to solve for:

1. Consuming domain events and persisting one or more query models.
2. Exposing the query model as an API.

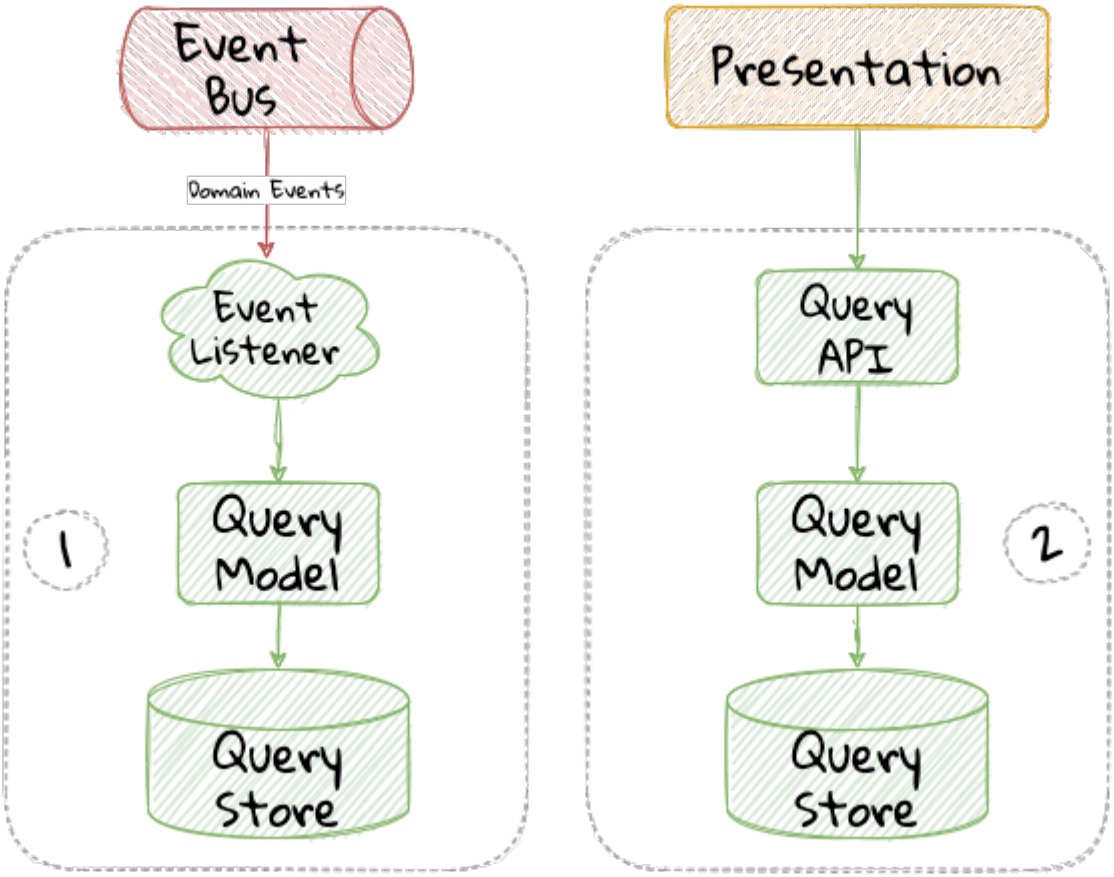

Figure 3. Query side dissected

Before we start implementing these concerns, let's identify the queries we need to implement for our LC issuance application.

Identifying queries

From the eventstorming session, we have the following queries to start with:

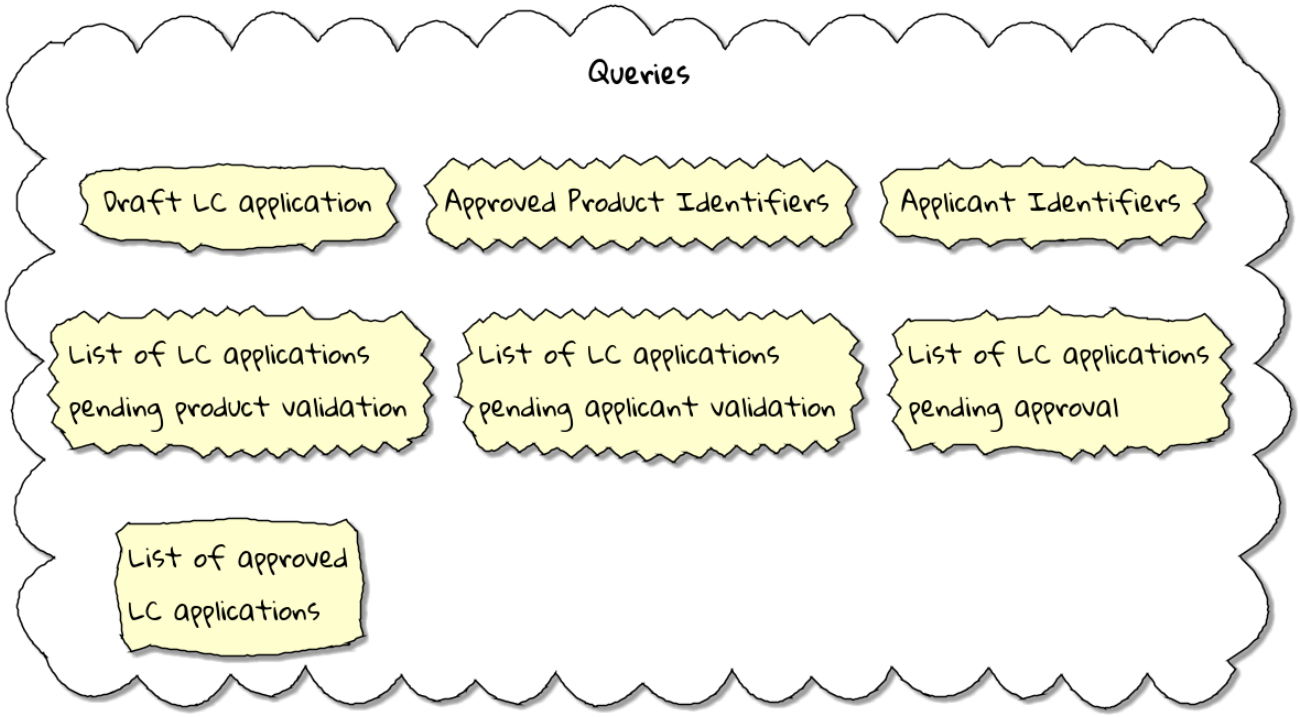

Figure 4. Identified queries

The queries marked in green, all require us to expose a collection of LCs in various states. To represent this, we can create an `LCView` as shown here:

The `LCView` class is an extremely simple object devoid of any logic.

```
public class LCView {

    private LCApplicationId id;
    private String applicantId;
    private String clientReference;
    private LCState state;

    // Getters and setters omitted for brevity
}
```

These query models are an absolute necessity to implement basic functionality dictated by business requirements. But it is possible and very likely that we will need additional query models as the system requirements evolve. We will enhance our application to support these queries as and when the need arises.

Creating the query model

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;
import org.springframework.stereotype.Component;

@Component
class LCApplicationStartedEventHandler {

    @EventHandler                                          ①
    public void on(LCApplicationStartedEvent event) {
        LCView view = new LCView(event.getId(),
                        event.getApplicantId(),
                        event.getClientReference(),
                        event.getState());                 ②
        // Perform any transformations to optimize access
        repository.save(view);                             ③
    }
}
```

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.

③ We finally save the view into an appropriate query store. When persisting this data, we should consider storing it in a form that is optimized for data access. In other words, we want to reduce as much complexity and cognitive load when querying this data.

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.

Query side persistence choices

Segregating the query side this way enables us to choose a persistence technology most appropriate for the problem being solved on the query side. For example, if extreme performance and simple filtering criteria are prime, it may be prudent to choose an in-memory store like Redis or MemCached. If complex search criteria and large datasets are to be supported, then we may want to consider something like ElasticSearch. Or we may even simply choose to stick with just a relational database. The point we would like to emphasize is that employing CQRS affords a level of flexibility that was previously not available to us.

Consuming the query model

Applicants usually like to view the LCs they created, specifically those in the draft state. Let's look at how we can implement this functionality. Let's start by defining a simple object to capture the query criteria:

```
import org.springframework.data.domain.Pageable;

public class MyLCsQuery {

    private String applicantId;
    private LCState state;
    private Pageable page;

    // Getters and setters omitted for brevity
}
```

Let's implement the query to retrieve the results for these criteria:

```
import org.axonframework.queryhandling.QueryHandler;

public interface LCViewRepository extends JpaRepository<LCView, LCApplicationId> {

    Page<LCView> findByApplicantIdAndState(          ①
            String applicantId,
            LCState state,
            Pageable page);

    @QueryHandler                                    ②
    default Page<LCView> on(MyDraftLCsQuery query) {
        return findByApplicantIdAndState(            ③
                query.getApplicantId(),
                LCState.DRAFT,
                query.getPage());
    }
}
```

① This is the dynamic spring data finder method.

② The `@QueryHandler` annotation provided by Axon framework routes query requests to the respective handler.

③ Finally, we invoke the finder method to allow to return results.

To connect this to the UI, we add a new method in the `BackendService` (originally introduced in Chapter 6) to invoke the query as shown here:

```
import org.axonframework.queryhandling.QueryGateway;

public class BackendService {

    private final QueryGateway queryGateway;                    ①

    public List<LCView> findMyDraftLCs(String applicantId) {
        return queryGateway.query(                              ②
                new MyDraftLCsQuery(applicantId),
                        ResponseTypes.multipleInstancesOf(LCView.class))
                .join();

    }
}
```

① The Axon framework provides the `QueryGateway` convenience that allows us to invoke the query. For more details on how to use the `QueryGateway`, please refer to the Axon framework documentation.

② We execute the query using the `MyDraftLCsQuery` object.

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