Table of Contents

Implementing Queries

> 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:

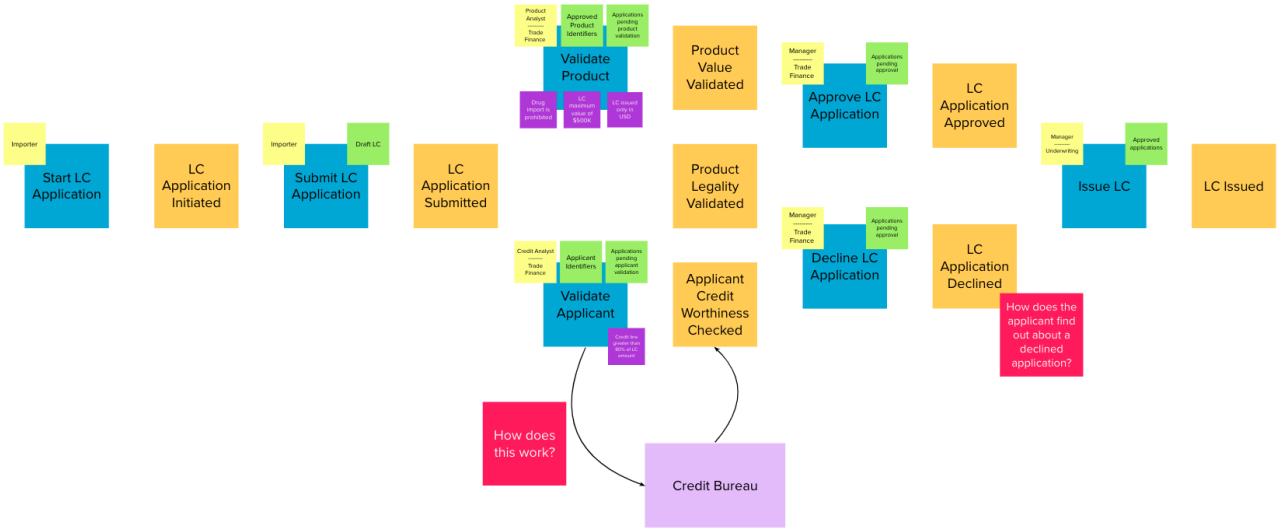

Figure 1. Recap of eventstorming session

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 "When to use CQRS" section in Chapter 3. 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). Let's look at what it means to apply CQRS in practical terms for the query side.

Implementing the query side

In Chapter 5, we examined 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:

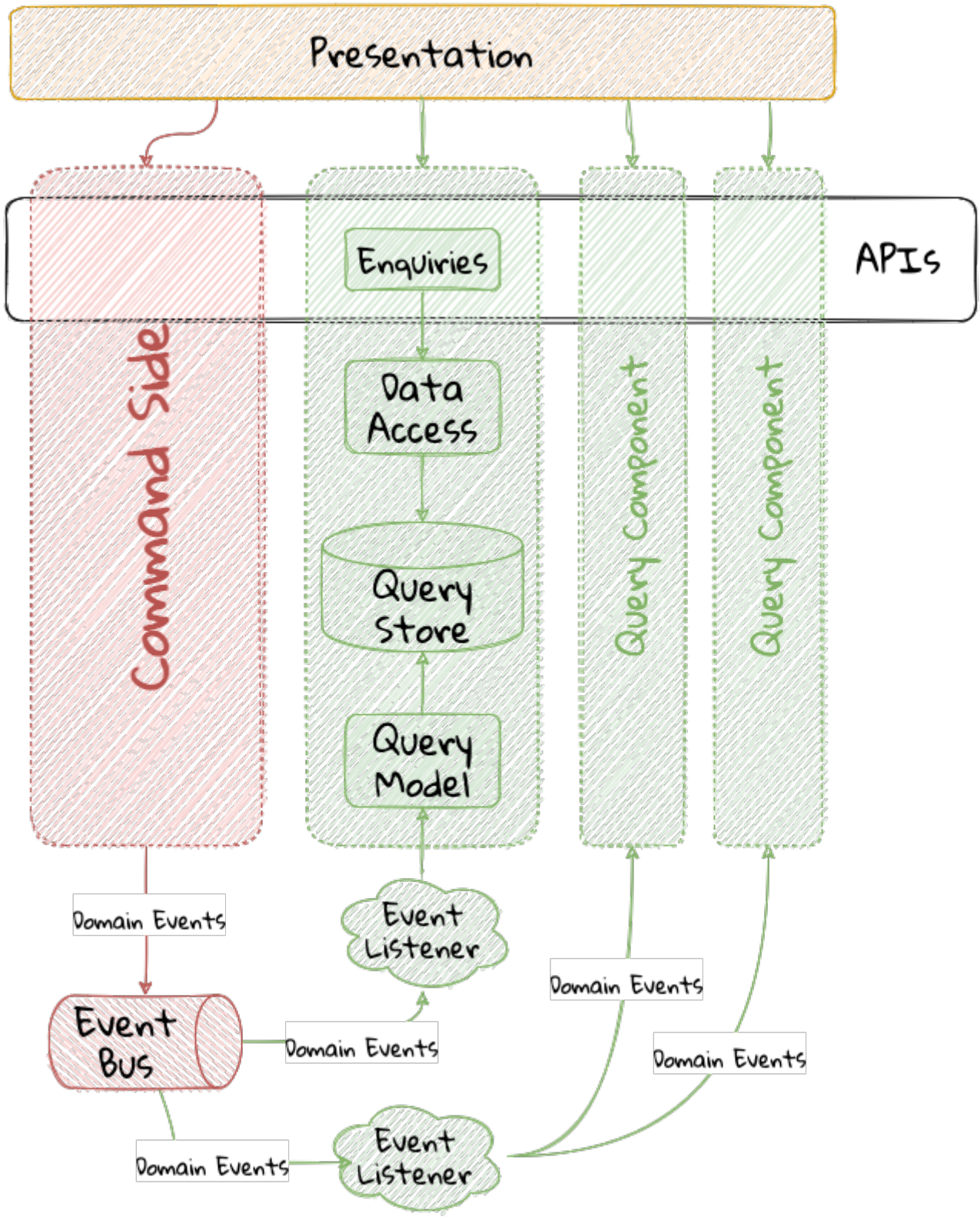

Figure 2. CQRS application — query side

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.

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.

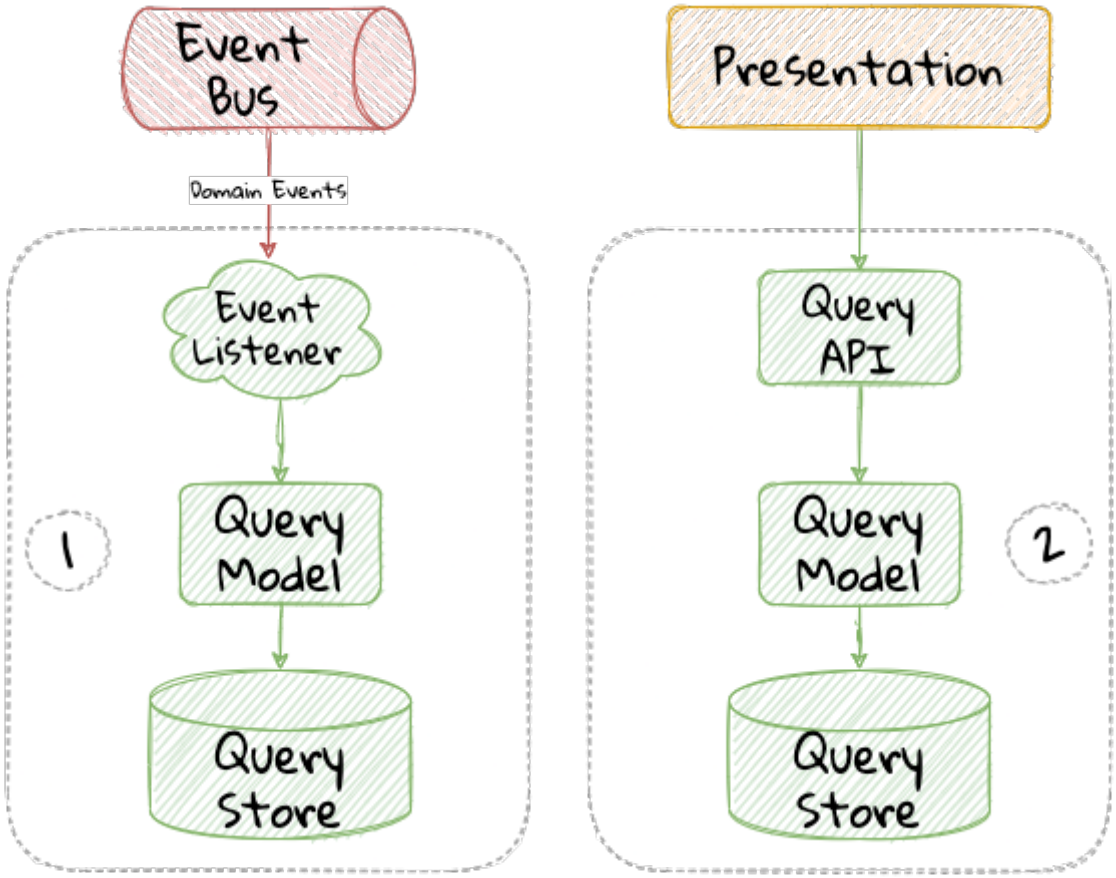

Figure 3. Query side dissected

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:

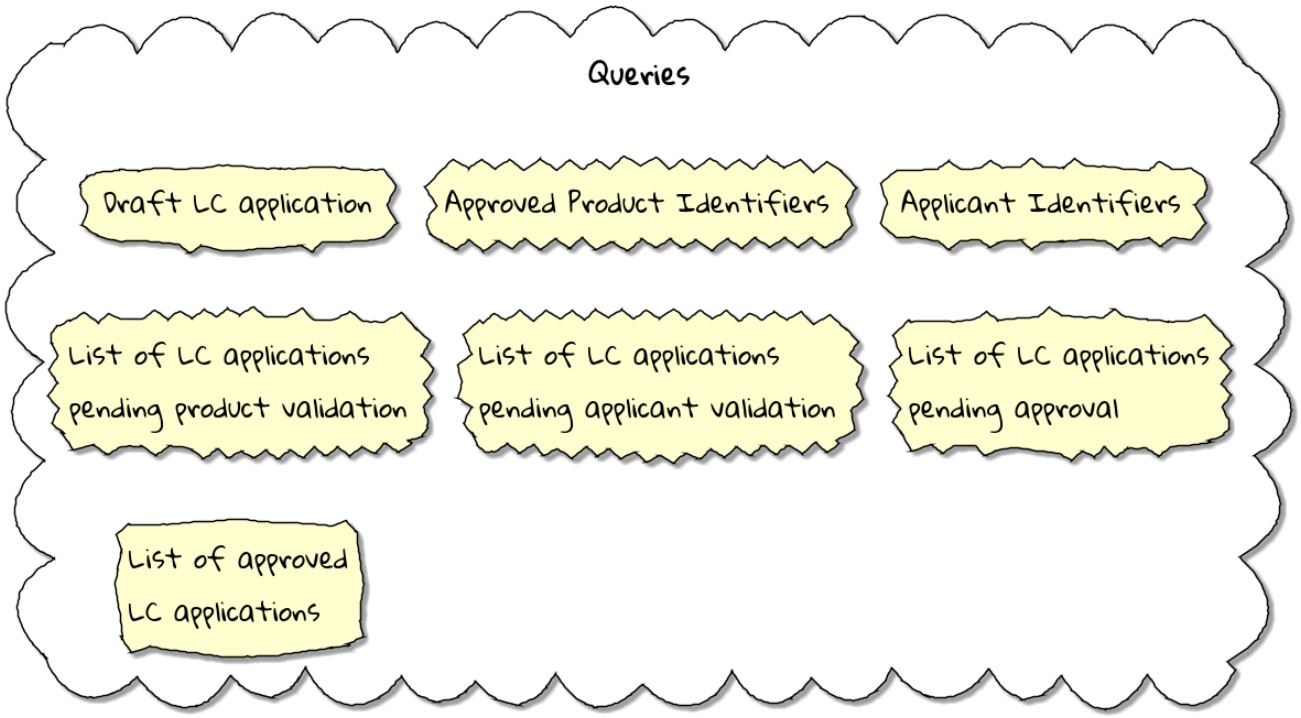

Figure 4. Identified queries

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 query model 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/analytics requirements 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.

Exposing a query API

Applicants 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 MyDraftLCsQuery {

    private ApplicantId applicantId;
    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 we will use to query the database.

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

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

What we looked at above, is an example of a very simple query implementation where we have a single `@QueryHandler` to service the query results. This implementation returns results as a one-time

fetch. Let's look at more complex query scenarios.

Advanced query scenarios

Our focus currently is on active LC applications. Maintaining issued LCs happens in a different bounded context of the system. Consider a scenario where we need to provide a consolidated view of currently active LC applications and issued LCs. In such a scenario, it is necessary to obtain this information by querying two distinct sources (ideally in parallel) — commonly referred to as the scatter-gather[1] pattern. Please refer to the section on scatter-gather queries in the Axon framework documentation for more details.

In other cases, we may want to remain up to date on dynamically changing data. For example, consider a real-time stock ticker application tracking price changes. One way to implement this is by polling for price changes. A more efficient way to do this is to push price changes as and when they occur — commonly referred to as the publish-subscribe[2] pattern. Please refer to the section on subscription queries in the Axon framework documentation for more details.

Historic event replays

The example we have looked at thus far allows us to listen to events as they occur. Consider a scenario where we need to build a new query from historic events to satisfy an unanticipated new requirement. This new requirement may necessitate the need to create a new query model or in a more extreme case, a completely new bounded context. Another scenario might be when we may need to correct a bug in the way we had built an existing query model and now need to recreate it from scratch. Given that we have a record of all events that have transpired in the event store, we can use replay events to enable us to construct both new and/or correct existing query models with relative ease.

We have used the term *event replay* in the context of reconstituting state of event-sourced aggregate instances. The event replay mentioned here, although similar in concept, is still very different. In the case of domain object event replay, we work with a single aggregate root instance and only load events for that one instance. In this case though, we will likely work with events that span more than one aggregate.

Let's look at how the different types of replays and how we can use each of them.

Types of replays

When replaying events, there are at least two types of replays depending on the requirements we need to meet. Let's look at each type in turn:

- **Full event replay** is one where we replay all the events in the event store. This can be used in a scenario where we need to support a completely new bounded context which is dependent on this sub-domain. This can also be used in cases where we need to support a completely new query model or reconstruct an existing, erroneously built query model. Depending on the number of events in the event store, this can be a fairly long and complex process.
- **Partial/Adhoc event replay** is one where we need to replay all the events on a subset of

aggregate instances or a subset of events on all aggregate instances or a combination of both. When working with partial event replays, we will need to specify filtering criteria to select subsets of aggregate instances and events. This means that the event store needs to have the flexibility to support these use cases. Using specialized event store solutions (like Axon Server[3] and EventStoreDB[4] to name a few) can be extremely beneficial.

Event replay considerations

The ability to replay events and create new query models can be invaluable. However, like everything else, there are considerations that we need to keep in mind when working with replays. Let's examine some of these in more detail:

Event store design

As mentioned in Chapter 5, when working with event-sourced aggregates, we persist immutable events in the persistence store. The primary use-cases that we need to support are:

1. Provide consistent and predictable **write** performance when acting as an append-only store.
2. Provide consistent and predictable **read** performance when querying for events using the aggregate identifier.

However, replays (especially partial/adhoc) require the event store to support much richer querying capabilities. Consider a scenario where we found an issue where the amount is incorrectly reported for LCs that were approved during a certain time period only for a certain currency. To fix this issue, we need to:

1. Identify affected LCs from the event store.
2. Fix the issue in the application.
3. Reset the query store for these affected aggregates
4. Do a replay of a subset of events for the affected aggregates and reconstruct the query model.

Identifying affected aggregates from the event store can be tricky if we don't support querying capabilities that allow us to introspect the event payload. Even if this kind of adhoc querying were to be supported, these queries can adversely impact command handling performance of the event store. One of the primary reasons to employ CQRS was to make use of query-side stores for such complex read scenarios.

Event replays seem to introduce a chicken and egg problem where the query store has an issue which can only be corrected by querying the event store. A few options to mitigate this issue are discussed here:

- **General purpose store**: Choose an event store that offers predictable performance for both scenarios (command handling and replay querying).
- **Built-in datastore replication**: Make use of read replicas for event replay querying
- **Distinct datastores**: Make use of two distinct data stores to solve each problem on its own (for example, use a relational database/key-value store for command handling and a search-optimized document store for event replay querying).

Do note that the **distinct datastores** approach for replays is used to satisfy an operational problem as opposed to query-side business use-cases discussed earlier in this chapter. Arguably, it is more complex because the technology team on the command side has to be equipped to maintain more than one database technology.

Event design

Event replays are required to reconstitute state from an event stream. In this article on what it means to be event-driven[5], Martin Fowler talks about three different styles of events. If we employ the *event carried state transfer* approach (in Martin's article) to reconstitute state, it might require us to only replay the latest event for a given aggregate, as opposed to replaying all the events for that aggregate in order of occurrence. While this may seem convenient, it also has its downsides:

- All events may now require to carry a lot of additional information that may not be relevant to that event. Assembling all this information when publishing the event can add to the cognitive complexity on the command side.
- The amount of data that needs to be stored and flow through the wire can increase drastically.
- On the query side, it can increase cognitive complexity when understanding the structure of the event and processing it.

In a lot of ways, this leads back to the CRUD-based vs task-based approach for APIs discussed in Chapter 5. Our general preference is to design events with as lean a payload as possible. However, your experiences may be different depending on your specific problem or situation.

Application availability

In an event-driven system, it is common to accumulate an extremely large number of events over a period of time, even in a relatively simple application. Replaying a large number of events can be time-consuming. Let's look at the mechanics of how replays typically work:

1. We suspend listening to new events in preparation for a replay.
2. Clear the query store for impacted aggregates.
3. Start an event replay for impacted aggregates.
4. Resume listening to new events after replay is complete.

Based on the above, while the replay is running (step 3 above), we may not be able to provide reliable answers to queries that are impacted by the replay. This obviously has an impact on application availability. When using event replays, care needs to be taken to ensure that SLOs (service level objectives) are continued to be met.

Event handlers with side effects

When replaying events, we re-trigger event handlers either to fix logic that was previously erroneous or to support new functionality. Invoking most (if not all) event handlers usually results in some sort of side effect (for example, update a query store). This means that some event handlers may not be running for the first time. To prevent unwanted side effects, it is important to undo the effects of having invoked these event handlers previously or code event handlers in an idempotent

manner (for example, by using an *upsert* instead of a simple insert or an update). The effect of some event handlers can be hard (if not impossible) to undo (for example, invoking a command, sending an email or SMS). In such cases, it might be required to mark such event handlers as being ineligible to run during replay. When using the Axon framework, this is fairly simple to do:

```
import org.axonframework.eventhandling.DisallowReplay;

class LCApplicationEventHandlers {
    @EventHandler
    @DisallowReplay  ①
    public void on(CardIssuedEvent event) {
        // Behavior that we don't want replayed
    }
}
```

① The `@DisallowReplay` (or its counterpart `@AllowReplay`) can be used to explicitly mark event handlers ineligible to run during replay.

Events as an API

In an event-sourced system where events are persisted instead of domain state, it is natural for the structure of events to evolve over a period of time. Consider an example of an `BeneficiaryInformationChangedEvent` that has evolved over a period of time as shown here:

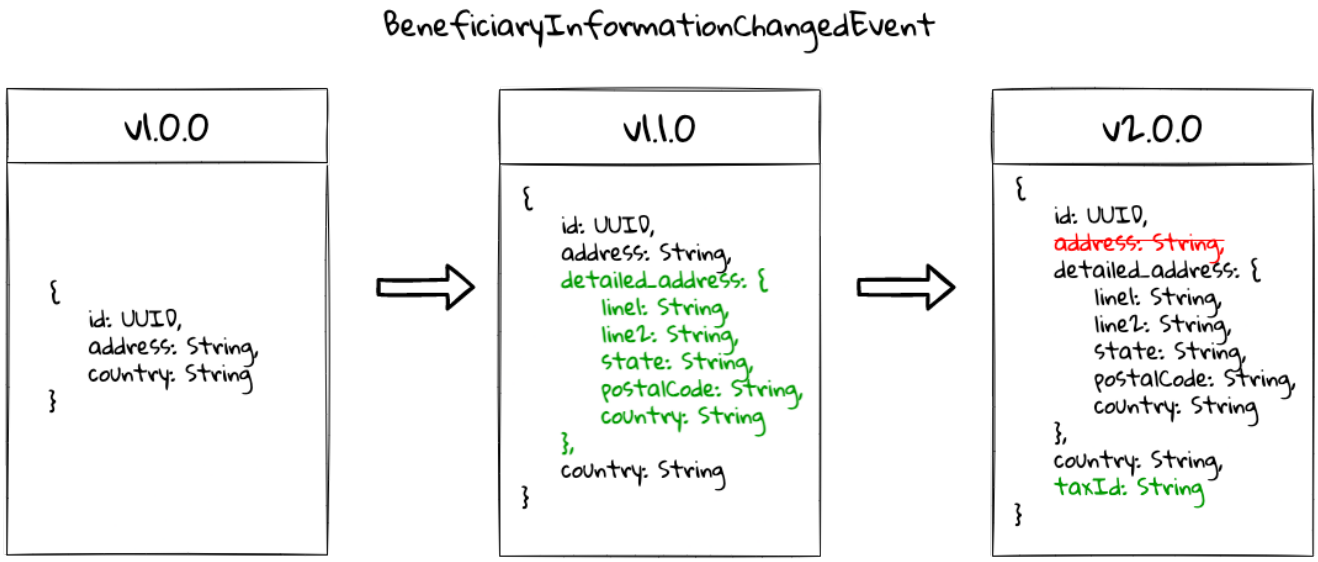

Figure 5. Event evolution

Given that the event store is immutable, it is conceivable that we may have one or more combinations of these event versions for a given LC. This can present a number of decisions we will need to make when performing an event replay:

- The producer can simply provide the historic event as it exists in the event store and allow consumers to deal with resolving how to deal with older versions of the event.
- The producer can upgrade older versions of events to the latest version before exposing it to the consumer.

- Allow the consumer to specify an explicit version of the event that they are able to work with and upgrade it to that version before exposing it to the consumer.
- Migrate the events in the event store to the latest version as evolutions occur. This may not be feasible given the immutable promise of events in the event store.

Which approach you choose really depends on your specific context and the maturity of the producer/consumer ecosystem. The axon framework makes provisions for a process they call **event upcasting**[6] that allows upgrading events just-in-time before they are consumed. Please refer to the Axon framework documentation for more details.

In an event-driven system, events are your API. This means that you will need to apply the same rigor that one applies to APIs when making lifecycle management decisions (for example, versioning, deprecation, backwards compatibility, etc.).

Summary

In this chapter we examined how to implement the query side of a CQRS-based system. We looked at how domain events can be consumed in real-time to construct materialized views that can be used to service query APIs. We looked at the different query types that can be used to efficiently access the underlying query models. We rounded off by looking at persistence options for the query side.

Finally, we looked at historic event replays and how it can be used to correct errors or introduce new functionality in an event-driven system.

This chapter should give you a good idea of how to build and evolve the query side of a CQRS-based system to meet changing business requirements while retaining all the business logic on the command side.

In this chapter, we looked at how to consume events in a stateless manner (where no two event handlers have knowledge of each other's existence). In the next chapter, we will continue to look at how to consume events, but this time in a stateful manner in the form of long-running user transactions (also known as sagas).

Questions

- In your context, are you segregating commands and queries (even if the segregation is logical)?
- What read/query models are you able to come up with?
- What do you do if you build a query model, and it turns out to be wrong?

[1] https://www.enterpriseintegrationpatterns.com/BroadcastAggregate.html

[2] https://www.enterpriseintegrationpatterns.com/PublishSubscribeChannel.html

[3] https://axoniq.io/product-overview/axon

[4] https://www.eventstore.com/eventstoredb

[5] https://martinfowler.com/articles/201701-event-driven.html

[6] https://docs.axoniq.io/reference-guide/axon-framework/events/event-versioning#event-upcasting