Axon Training – Labs

Fundamentals I

1. Introduction

This document describes the labs of the Axon training. The purpose of these labs is to give the participants the possibility to apply the theory using scenarios like those in the real world.

The labs are based on the sections covered by the training. Your trainer will provide the instructions about which lab to execute at what moment in time. Be aware that the majority of lab effort will be performed outside the scheduled presentation meetings with the trainer.

Labs may provide background information. This is usually information required to execute the lab and provides the context in which the lab should be executed. In case something is not clear, don't hesitate to ask the trainer for more information.

A communication channel has been made available to the candidates. The purpose of this channel is to discuss approaches for the exercises with your peers. Make sure to contact the trainer if you *do not* have access to this channel yet.

Hints provide information to the candidates with some guidelines and common pitfalls.

If you wish to contact us after the training, you can direct your questions, comments and concerns to AxonIQ via info@axoniq.io, or visit axoniq.io for contact details.

Have fun!

The Domain

The scope of the labs is an application that is used by a hotel chain to allow administration of Hotel Gift Cards. These Hotel Gift Cards are *issued* (individually or in bulk) and each have a unique identifier. Each Hotel Gift Card is assigned an amount when issued, which can be *redeemed*, either at once, or in parts. Transactions may also be *reimbursed*, when for example the customer canceled their hotel booking.

The Labs

A baseline has been prepared for each of the labs. Each of the labs can be executed against its baseline, available as a submodule in the root maven project.

Need help?

If you're stuck, you can discuss what to do on the provided communication channel, or peek into the baseline for the next lab, as it contains the solution of the lab you're working on. Lastly, you can also check the solutions projects. The trainer will chip in on the communication channel at best effort, especially outside of office hours (CET/CEST).

Build

When building the project with maven, by default it will only build the labs projects. To build an individual lab, use `-Plab#` (for example: `-Plab2`). To build all modules, including the solutions, use `-Pall`.

Module 1 - Core API Design

Before starting a CQRS based project, it is wise to briefly think about the types of messages published. In other words, we need to identify Commands, Events and Queries, together with their responses. After this, a rough sketch can be made of the UI to implement.

Some of you will be asked to share their derived message with the other candidates. Choose whichever drawing tool you prefer, on or offline, as long as it can be shared.

Exercise 1

We will focus on the Gift Card aspect of our Domain. Identify the Commands that can be sent and the Events that may be published as a result of them. Also think about the error-cases when handling a Command. What should happen when a Command is sent while the state of the Aggregate doesn't support it?

Hint

There are many ways to "design" messages. A commonly used technique is Event Storming, where post-it notes of different colors are used to represent the different types of messages.

Alternatively, you can use a notation like UML Sequence Diagrams. Write the entity name top center, with a vertical line underneath. To the left of the line, arrows towards the line represent commands. Arrows to the right of the line (pointing outwards) represent the events published as a result.

Exercise 2

The application also has a UI. In this UI, the user would like to see a Summary of the cards. Which Queries would the UI need to send? What type of view model would be needed to support the UI? What will the UI look like knowing all this information?

Module 2 - Command Model

In this lab, you're going to implement the Command Model of the Hotel Gift Card Admin application. It will accept incoming commands, update a model and produce Events as a result. In turn, these events will be used to (re)create the Command Modal, through Event Sourcing.

Exercise 1

Find the `GiftCard` Aggregate in the "Lab2" module. As you can see it is an empty class. The `coreapi` package contains several Java files, which define the structure of the Commands and Events you'll need.

Implement the Command Handlers of the `IssueCardCommand`, which creates a new `GiftCard` instance, the `RedeemCardCommand`, which is used when a user spends part of the `GiftCard`, and the `ReimburseCardCommand`, which can be used to reimburse a previous Transaction. Take the error-cases form Lab 1 into account too, as those are part of the decision making logic.

Hint

Note that the `IssueCardCommand` should create a new `GiftCard`. To do that, place the `@CommandHandler` annotation on a Constructor of `GiftCard`, which accepts an `IssueCardCommand` as parameter. The other commands should act on existing instances of a `GiftCard`.

Don't forget putting the `@Aggregate` annotation on the `GiftCard` class, otherwise Axon will not recognize it as such. And remember, an aggregate does not necessarily consist of a single class.

Exercise 2

Now that the decision making of our application has been implemented, we can start introducing the state changes. refactoring the `GiftCard` aggregate. Find which events contain valuable information to support our decision making. Add `@EventSourcingHandler` methods for these events and set the required state in them for the `GiftCard` aggregate.

Exercise 3

One of the advantages of Event Sourcing is that we can use Commands and Events to describe the expected behavior of an aggregate. Check the `GiftCardTest` class in the test sources. It defines several test scenarios which are hard-coded to fail. Use the fixture (which has been initialized for you) to describe the expected behavior.

Run the tests and continue refactoring the `GiftCard` aggregate until they pass.

Note

Axon uses a mechanism to detect "illegal" state changes and depends on an equality check. `List`s are only considered equal when all their elements are. Implement `equals()` and `hashCode()` on the Command Model entities and run the tests again.

Launch

Firstly, we need to start up Axon Server. You can download it from https://download.axoniq.io/training/AxonServer.zip and start it by running "`java -jar axonserver.jar`". If you prefer to use Docker, you can run the the following to achieve the same:

```
docker run -d --name axonserver -p 8024:8024 -p 8124:8124
-e AXONIQ_AXONSERVER_DEVMODE_ENABLED=true axoniq/axonserver
```

After that, start the application and visit http://localhost:8080. The UI should render and allow you to issue, redeem and reimburse cards. Check out the events stored in Axon Server by visiting http://localhost:8024 and checking out the 'Search' page.

Module 3 - Event Handling & Projections

In this Lab, we're going to handle the emitted events from the `GiftCard` aggregate in an Event Handler, and create/update a view model.

Exercise 1

The application now correctly emits Events when Gift Cards are issued, redeemed or reimbursed. It is now time to give the user an overview of issued gift cards.

Check the `CardSummary` class. It contains a few getter methods that return basic (and wrong) values. This is done to make the code for the UI compile, as it depends on these methods.

Evolve the `CardSummary` class into a proper JPA Entity. Think twice before generating getters and setters; what DDD concepts can you use here?

Exercise 2

Now that our Query Model is up to par, we can start updating it. Find the `CardSummaryProjection`. Add `@EventHandler` annotated methods for the three types of events defined in the core API.

Further implement these methods to create and update the `CardSummary` instance. A `CardSummaryRepository` is already injected for you, which you can use to find and save `CardSummary` instances.

Exercise 3

The view model is created and automatically updated by events. However, we still need to get this information to the user interface. For this, we are going to explore explicit Query Messages in this exercise.

The dispatching logic has been provided, so you only need to correctly implement the `@QueryHandler` annotated methods in the `CardSummaryProjection` class.

Extra

The user interface has already been wired to send these query messages. If you want to know how, check out the `CardSummaryDataProvider` class. It implements the logic needed by Vaadin to create a scrollable grid that lazily retrieves data as the user scrolls up and down the grid.

Launch

Run the application again. Now, you should be able to see the grid represent your changes as you send the commands.

Module 4 - Sagas and Deadlines

The application needs to be connected to the booking system, which we're going to simulate in this lab. There is an extra panel in the User Interface that allows you to trigger an Event from this system. When a booking is placed, which needs to be paid with a Gift Card, the Gift Card system must redeem the relevant card and the Booking system must be notified of the result.

Exercise 1

Check the `BookingSaga`. It is completely empty. Implement the necessary `@SagaEventHandler` methods.

Hint

Which association properties will you use? Note that the "`bookingId`" that can be used to trigger a new Saga is not available in the Hotel Gift Card application. And rightfully so. Don't forget to associate the Saga with the correct properties before sending out commands to redeem funds from a gift card.

Exercise 2

When placing a booking, users can decide to make a partial payment with a gift card by checking the "Partial Payment" checkbox. The rest of the payment has to be fulfilled in seven days, otherwise the booking should be rejected. Schedule a deadline with the `DeadlineManager` and add a `@DeadlineHandler` to implement this behaviour.

Exercise 3

To ensure our booking transaction logic works as desired, tests should be added. A `BookingSagaTest` class has been provided to this end with failing tests. Implement these test methods such that they succeed.

Launch

Launch the application and use the Booking panel to trigger a booking. You should see the effects in the grid on the bottom of the page. In some cases, you might have to manually refresh the grid.