Axon Training – Labs

Fundamentals II

Introduction

This document describes the labs of the Axon Fundamentals II 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 5 - Snapshotting and Event Processors

The application is functionally complete now. It is time to start tuning the non-functionals.

Unlike previous labs, the labs from here on use H2 with a persistent configuration. In other words, the query model isn't erased when restarting the application.

Exercise 1

It has shown that some clients of our application use a single hotel gift card very frequently. We should configure a `SnapshotTriggerDefinition` as this imposes strains on loading the `GiftCard` aggregate. An `EventCountSnapshotTriggerDefinition` will do in this case.

Either add a `SnapshotTriggerDefinition` bean and set it on the `@Aggregate` annotation or use the `AggregateConfigurer` directly to configure it.

Exercise 2

The business noticed users want additional information in the `CardSummary` grid: the number of transactions executed against a hotel gift card. Change the implementation of the Query Model and the Event Handlers to include this information.

In order to perform replays, but also to reliably process events asynchronously, we need to use a Tracking Event Processor. As this is the default in Axon we are already one step ahead to initiate a replay of our query model.

To be able to *only* target the `CardSummaryProjection` to recreate the `CardSummary` model, we need to assign it to a Processing Group. By doing so we can replay this projection, without affecting others.

Note

If you need to change the event processing configuration, there are two ways to go about. You can either use the `EventProcessingConfigurer` instance from the Configuration API or annotate the event handler with `@ProcessingGroup({your-processor-name})` and configure the processor in `application.properties` if needed.

Launch

Before starting the improved hotel gift card application, delete any H2 database files in your workspace root (`database.mv.db` and `database.lock.db`) to force a rebuild of the view models.

Launch the application again; note that the column is already properly updated with information from activity that happened in the past. If it isn't, make sure the database files were properly deleted before starting the application.

Exercise 3

You would never delete a database in a production environment to *force* a replay. Instead, you should use the Reset API to reset your Tracking Event Processors to start a conscious replay.

On a reset we need to clear out the `CardSummary` table to allow it to be rebuilt. Add an `@ResetHandler` annotated method to the `CardSummaryProjection` which clears out the `CardSummary` table.

Additionally, we need to be able to trigger this reset from the UI. To that end, a `ManagementEndpoint` is provided in the `gui` directory. The only thing left is to retrieve the correct Tracking Event Processor from the `EventProcessingConfiguration` and to reset it when this reset-endpoint is hit.

Module 6 - Preparing for production

Our application is evolving and adoption in the organization is increasing. Due to this a couple of adjustments need to be made to the hotel gift card application.

Exercise 1

We want to add logging to our set up for each message going in and out of the application. Configuring a `LoggingInterceptor` as a Message Handler Interceptor and Dispatch Interceptor is the fastest way to achieve this.

Create a single `LoggingInterceptor` bean and configure it on the `CommandBus`, `EventBus`, `EventProcessors` and `QueryBus`. Remember that fine-grained configuration for `EventProcessors` goes through the `EventProcessingConfigurer`.

Bonus

You can also implement your own `MessageHandlerInterceptor` / `MessageDispatchInterceptor` and configure it. When doing so, recall that the implementation has generics for the type of `Message` to act on.

Exercise 2

The default XML format of the events is rather big, making it so that the event store increases quicker than expected. Configure the event serializer to be a `JacksonSerializer` to end up with more concise event formats. This also requires the addition of Jackson specific annotations to the event classes.

Note

Currently we have an event store filled with XML serialized events. Normally you would migrate the events when such a scenario occurs, as you would **never** remove production events. As this is not a production environment we will drop the old event store to save ourselves some time.

Launch

Start the hotel gift card application. Functionally everything should work the same as before. Test if this is the case and check whether the `LoggingInterceptor` is doing its job.

Exercise 3

Management would like to get more information about the Shop in which gift cards are issued. Therefore, the `CardIssuedEvent` should carry the `shopId`

where it was issued. One way to accommodate for changes in the structure of events, is by using upcasters.

Add a shopId property of type String to the `CardIssuedEvent` and `IssueCardCommand`. This will break the existing code. That's no problem. Just try to compile and fix all errors. Also, be mindful of the `CardSummary` class. Because we are changing the structure of the `CardIssuedEvent`, we need to add an `@Revision`. Choose revision "1" for this new one.

Exercise 4

It is now time to write the upcaster. Create a (Spring) `@Component` annotated class that extends `SingleEventUpcaster`. It has two abstract methods that need to be implemented.

Implement the upcaster to add a text node called "`shopId`" with value "Unknown" (which has been decided to represent the Unknown shop).

Hint

Writing an upcaster for Jackson serialized events requires the usages of JsonNode as the expected representation type. This type, however, does not allow the addition of nodes

The upcasters are automatically wired and passed to the `AxonServerEventStore` (or any other Event Store) through the Spring auto-configuration process. If you have multiple Upcaster implementations, you can use Spring's `@Order` annotation on the upcasters to ensure they are injected in the correct order.

Launch

Before launching, clear the database (not the event store). When restarting, the Events will be replayed to your handlers, and you should see "Unknown" as shopId for previously issued cards. Newly issued cards get the shopId provided in the form.

Food for thought

Besides using upcasters, what could be another way to retain compatibility?

Module 7 - CQRS & Distributed Systems

A single instance of the application will not do the job. Also, all components are currently included in a single, monolithic deployment. In this lab, we are going to decompose the application into separately deployable units. Also, we are going to separate our application into two bounded contexts - `giftcard` and `booking`.

Instead of creating physically different jar files, we will be using Spring Profiles to enable or disable certain beans. Those profiles are:

- *`command`* - runs the command part of the application only (i.e. the `GiftCard` aggregate)
- *`query`* - runs our projections
- *`saga`* - coordinates communication between two contexts: `giftcard` and `booking`
- *`booking`* - runs the `BookingCommandHandler`

We are going to run an application per profile. On top of that, we are going to run an instance without profiles for UI.

Exercise 1

Let's segregate our application in the `giftcard` and `booking` contexts. Since the *`saga`* is responsible for coordination between those two contexts it'll subscribe to both of them. The other three applications will run in their own contexts: *`command`* and *`query`* will be part of the `giftcard` context and *`booking`* belongs to the in `booking` context.

Take a look at the `configureSagaProcessor` method in the `HotelGiftCardApplication` class to check how to tie the `saga` profile to both contexts.

Exercise 2

Now, it's time to run Axon Server in clustered mode. This can be achieved in two ways: using the Axon Server CLI or through the "autoclustering" feature. We are opting for the second approach, since it is sufficient and straightforward.

`Lab7` contains three additional directories: *`axonserver1`*, *`axonserver2`* and *`axonserver3`*. Put the *`axonserver.jar`* of Axon Server Enterprise in all three, together with a copy of the *`axoniq.license`* file. The directories already contain configuration properties for the corresponding Axon Server

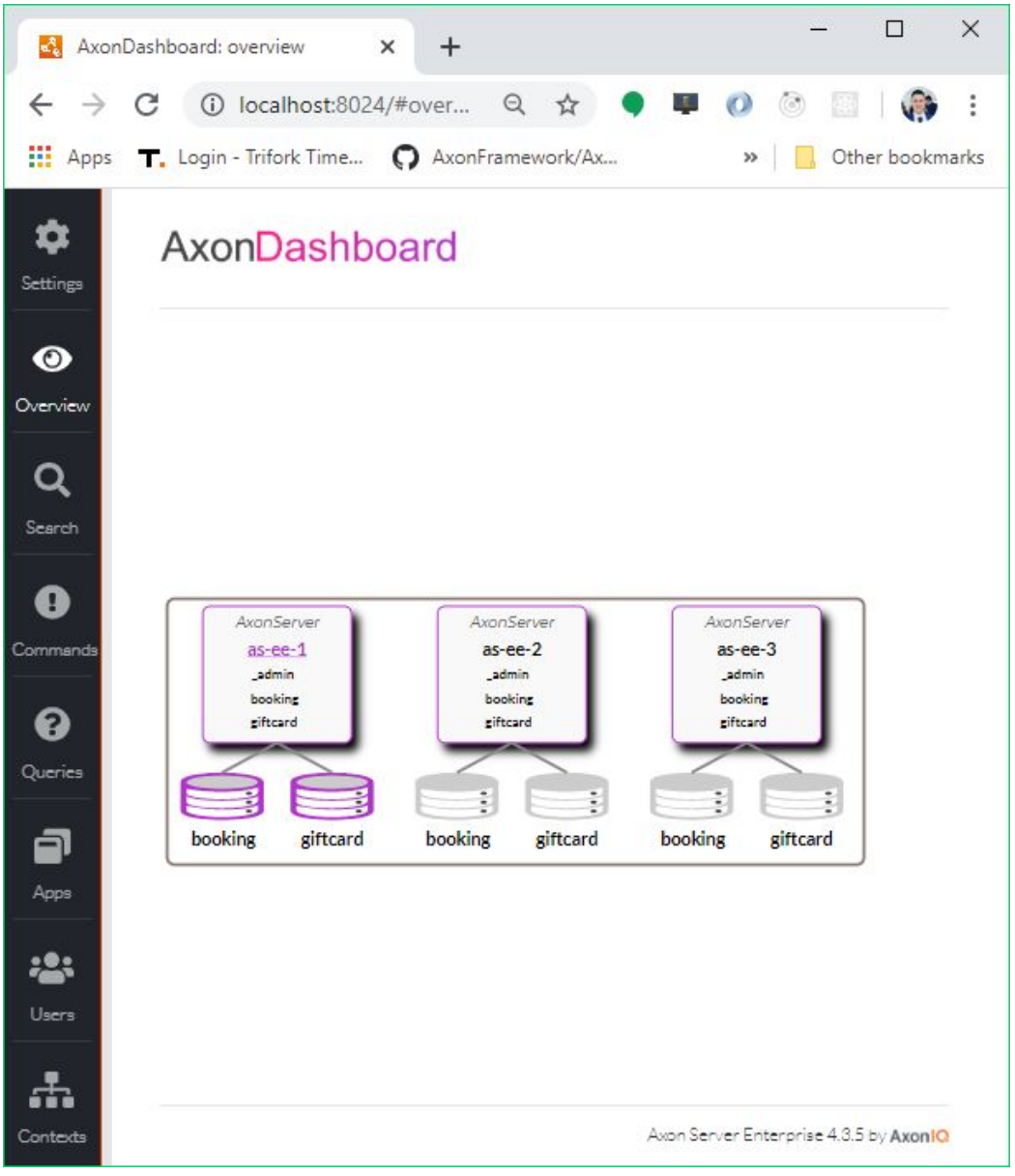
nodes. In these configuration properties you can spot specifics for hosts and names, but also configuration to enable autoclustering:

```
axoniq.axonserver.autocluster.first=localhost:8224
```

```
axoniq.axonserver.autocluster.contexts=_admin,booking,giftcard
```

This is the same for all nodes and indicates their respective contexts to join and which node is going to be the first in the cluster.

Run the Axon Server nodes using "`java -jar axonserver.jar`" command. You should be able to see something like this when visiting http://locahost:8024 (i.e. the Axon Server Dashboard):

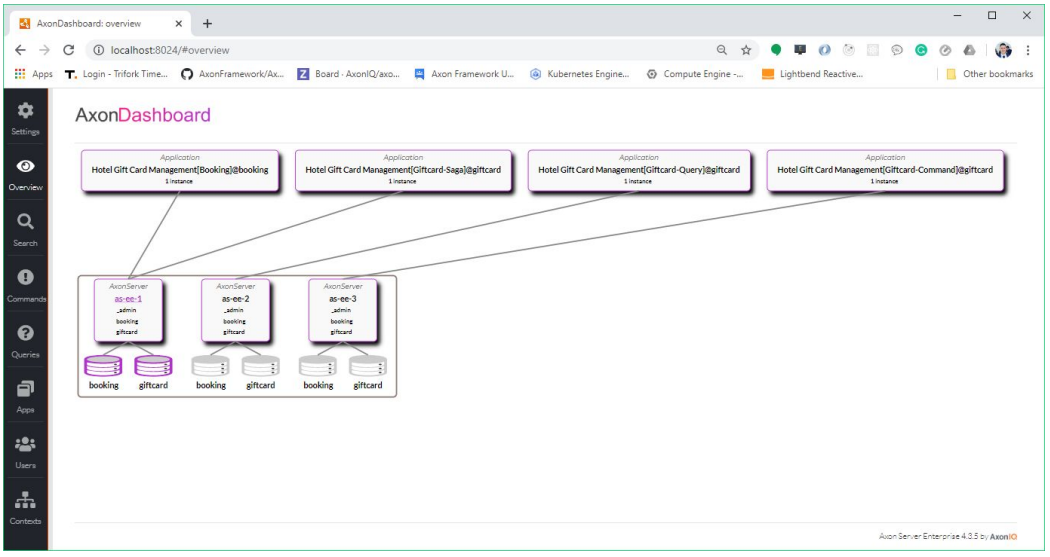


Exercise 3

It is time to start our applications. Create an *app* directory which will be used for runtime purposes of our segregated `giftcard` applications. Build `lab7` and copy the jar from the target directory to the newly created *app* directory and rename it to *`app.jar`*. In the `lab7` resources directory you will find the *`application.properties`* files per profile; copy them as well to the *app* directory. Let us use the following commands to start the applications:

- "`java -jar app.jar`" will start the application with no profiles. It'll contain the UI logic only.
- "`java -jar -Dspring.profiles.active=booking app.jar`" - This starts the `booking` application in `booking` context.
- "`java -jar -Dspring.profiles.active=command app.jar`" - This starts the application `command` profile in the `giftcard` context responsible for dealing with our Command Model.
- "`java -jar -Dspring.profiles.active=query app.jar`" - This will start the `query` application in the `giftcard` context responsible for dealing with our Query Model.
- "`java -jar -Dspring.profiles.active=saga app.jar`" - This starts the `saga` application responsible for tying both two contexts together.

After starting all those applications, you should see the following outcome in the dashboard:



Bonus Exercise

Consider a case where (due to high availability patterns) your Axon Server nodes are not part of the same data centers. Added, your applications might be in different centers as well. It would be convenient to connect our applications to the Axon Server node which lives in the same data center. Fortunately, Axon provides the `tagging` feature for Axon Server and Framework applications to achieve just that.

Let us assume that `axonserver1` is located in Utrecht, `axonserver2` in New York and `axonserver3` runs in Novi Sad. All Axon Framework applications are situated in Utrecht except for the `query` service, which resides in New York, and the `booking` service, which runs in Novi Sad.

Configure the tags on the servers and applications accordingly.

If you have configured your tags correctly, the dashboard should display the following: