Labs

Axon Framework
Introduction and Advanced Features

Introduction

This document describes the labs part of the Advanced Features of Axon Framework training. The purpose of these labs it to give the participants the possibility to apply the theory using scenario's similar to 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.

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

Hints provide information to the candidates with some guidelines and common pitfalls. They do not provide any solutions.

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. For information about Axon Framework, visit www.axonframework.org.

Have fun!

Background

The scope of the labs is an application that is used by a teller in a shop, to allow administration of Gift Cards. These Gift Cards are issued (individually or in bulk) and each have a unique identifier. Each Gift Card is assigned an amount, which can be redeemed, either at once, or in parts. Transactions may also be reimbursed, when the goods that were bought are returned.

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.

If you're stuck, you can ask your trainer for help, or peak 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.

When building the project with maven, be 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`.

Preparation

For this lab, a basic Spring Boot web application has been prepared for you to build on. Go to https://download.axoniq.io/training/labs.zip to download the project.

The zip contains a number of files and folders. The labs folder contains the sources needed for each of the labs. It also contains the AxonDB jar, an executable Jar file together with a configuration file. This will be used from lab 7 onwards.

Lab 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.

This exercise is to be done in teams of 2 to 4 people, depending on the size of the class. After coming up with a design, each team will demonstrate its results to the other teams

Exercise 1 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. What should happen when a Command is sent while the state of the Aggregate doesn't support it?

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?

Hints 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 similar to 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.

Lab 2 Command Handling

In this lab, you're going to implement the Command Handling component of the Gift Card Admin application. It will accept incoming commands, update a model and produce Events as a result.

Exercise 1 Find the `GiftCard` Aggregate in the "Lab2" module. As you can see it is an empty class. The `coreapi` package contains two Kotlin 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.

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.

Exercise 2 Put the JPA annotation on the entities required to store the `GiftCard` in a relational database.

Note Note that Event Sourcing is not used yet. Some mappings might feel awkward and sometimes JPA will force you to put properties in an Entity that you did not anticipate. That's fine for now, we'll resolve that later.

Strictly speaking, you don't need to emit Events yet to make this lab work. You can decide for yourself if you do it, or not. If not, you'll have to do it in the next lab.

Hint The entities you'll be looking for are `@Entity`, `@ManyToOne`, `@OneToMany` (don't forget the "cascade all" setting) and `@Id`.

Launch Start the application and visit http://localhost:8080. The UI should render and allow you to issue, redeem and reimburse cards.

Lab 3 Event Handlers

In this Lab, we're going to emit events from the `GiftCard` aggregate, and handle them in an Event Handler.

Exercise 1 Ensure that all Command Handlers publish the appropriate events. You can use the static `AggregateLifecycle.apply()`, or pass the `EventBus` as a parameter to your `@CommandHandler` methods. When using the latter, use the `GenericEventMessage` to wrap your Events.

Exercise 2 Check the `CardSummaryProjection` class in the `query` package. It is empty, and only annotated with Spring's `@Component`. Add Event Handlers for the three types of Events defined in the core API. Don't worry about the implementation too much, just yet. A simple `System.out` (or slf4j logging) will do for now.

Launch Launch the application. Issue a few cards and check the console for logging. You should see log statements generated by the `CardSummaryProjection`.

Lab 4 Implement View Model

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.

Exercise 1 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 Find the `CardSummaryProjection` class. It has three `@EventHandler` annotated methods. These are probably similar to the ones you created in Lab 3. Implement these methods to update the `CardSummary` instance. An `EntityManager` 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. While we can use traditional approach, we're going to explore explicit Query Messages in this lab. Implement the `@QueryHandler` annotated methods.

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.

Lab 5 Event Sourcing

So far, our Gift Card's state is maintained in a relational database, while it emits Events to the bus. However, how can we guarantee that these Events truthfully represent the actual changes done? We can't. That's what Event Sourcing is for (among other things).

Exercise 1 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 a number of test scenarios while are hard-coded to fail. Use the fixture (which has been initialized for you) to describe the expected behavior. They should run, but expect them to fail.

Exercise 2 Now that the tests are defined, we can start refactoring the `GiftCard` aggregate. Find places where state is modified inside the `@CommandHandler` methods, and move these to the appropriate `@EventSourcingHandler` methods.

Run the tests and continue refactoring 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 `GiftCardTransaction` and run the tests again.

Launch Launch the application. Can you spot the difference?

Visit http://localhost:8080/h2-console to access the data. You should be able to see a DOMAIN_EVENT_ENTRY table. Have a look at the contents. Unfortunately, H2 renders binary data as hex values. You can use an online hex-to-text converter to read the content.

Lab 6 Saga

The application needs to be connected to the sales 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 an item is sold, which needs to be paid with a Gift Card, the Gift Card system must redeem the relevant card and the Sales system must be notified of the result.

Exercise 1 Check the `GiftCardPaymentSaga`. It is completely empty. Implement the necessary `@SagaEventHandler` methods.

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

Launch Launch the application and use the Sales panel to trigger a sale. 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.

Lab 7 Event Processors and Replays

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

Unlike previous labs, the labs from here on use H2 using a persistent configuration. In other words, data doesn't get lost when restarting. Also, the application is configured to use AxonDB as an Event Store. This allows you to delete the database (by deleting database* files).

Exercise 1 In order to be able to perform replays, but also to simply reliably process events asynchronously, we need to use a Tracking Processor.

Configure the `CardSummaryProjection` Event Handlers to be assigned to a Tracking Processor, instead of the default Subscribing Processor.

Hint There are two ways to go about. You can either use the `EventHandlingConfiguration` instance (use an `@Autowired` method in your Spring Configuration class to access it) to use the Configuration API, or annotate the event handler with `@ProcessingGroup("choose name here")` and configure the processor mode in `application.properties`.

Launch Launch AxonDB and the application and verify everything still works as expected.

Exercise 2 Now that we are able to replay events to our components, let's take advantage of that. The users have requested an extra column in the grid: the number of transactions executed against a Gift Card.

Change the implementation of the Event Handlers to include this information.

Before restarting, delete the database file again.

Note Note that in a production environment, you would not just delete a database. You would either clear specific tables and remove TOKEN_ENTRY rows owned by the Processor to be reset.

Launch Launch the application again, an note that the column is already properly updated with information from activity that happened in the past.

Lab 8 Upcasters

Our application is evolving and adoption in the organization is increasing. Management would like to get more information about the Shop in which gift cards are issued. Therefore, the `CardIssuedEvent`s should carry the `shopId` where it was issued.

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

Exercise 1 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. Because we are changing the structure of the `CardIssuedEvent`, we need to add an `@Revision`. Choose revision "1" for this new one.

Exercise 2 It is now time to write the upcaster. Create an (Spring) `@Component` annotated class (name doesn't matter much) that extends `SingleEventUpcaster`. It has two abstract methods that need to be implemented.

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

Note Dom4j has been added as a dependency. Since the default serializer (XStream) serializes to XML, it is nice to have a more friendly API to change the structure of the event.

The upcasters are automatically wired and passed to the `AxonDBEventStore` constructor in the `GiftCardApplication` class. 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.

Bonus Exercise The upcaster needs a test case to prove it works. Implement a test. You can create an `InitialEventRepresentation` and pass it a `GenericDomainEventEntry` that contains the payload. Most of the information in the `GenericDomainEventEntry` is actually not needed by the upcaster, so any value will do.

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?

Lab 9 Distributing the application

A single instance of the application will not do the job. Also, all components are included in a single, monolithic deployment. In this lab, we are going to decompose the application into separate parts.

Instead of creating physically different jar files, we will be using Spring Profiles to enable or disable certain beans. You can choose active profiles per instance in the launch screen of your IDE. By default, the "command", "query" and "saga" profiles are all enabled.

Exercise 1 The pom.xml file has been prepared to include JGroups and the JGroups based connector for the `DistributedCommandBus`. Enable the distributed command bus by changing `application.properties` and setting `axon.distributed.enabled=true`.

Note JGroups will automatically bind to an address, based on the network configuration. In our case, we want to bind to a local address, so we do not rely on public networks or interfere with our fellow class mates. Also, for the purpose of local deployments, we want to automatically launch the embedded Gossip Server. To do this, add the following to `application.properties`:

```
axon.distributed.jgroups.bind-addr=127.0.0.1
axon.distributed.jgroups.gossip.auto-start=true
```

Exercise 2 Next, we're going to publish some events to AMQP. Check the `AmqpConfig` class. It contains basic Spring AMQP configuration to declare Exchanges and Queues. It also defines a `SpringAMQPMessageSource`, which is the bridge between AMQP queues and Axon's Event Processors.

Enable publishing events to AMQP by adding the `axon.amqp.exchange` property to `application.properties`.

Note the `@ProcessingGroup` annotation on the `SystemOutHandler` class in `AmqpConfig`. Configure this processor to use the `SpringAMQPMessageSource` as its source (either using the Configuration API or `application.properties`).

Hint You will need to run RabbitMQ (or another AMQP message broker) for this exercise. The easiest way to do so is by running a Docker container.

Launch Launch the application. Try launching multiple instances in different configurations. What do you see happening?

Lab 10 Monitoring

Now that the application is distributed, it is becoming increasingly important to monitor the health of each of the components. Also, we want to be able to trace activity to the user interface session that it was performed in.

Exercise 1 A class called `VaadinSessionInterceptor` has been provided, which attaches the Vaadin Session ID to outgoing messages. We need to make sure that the interceptor is used when sending commands.

Override the default `CommandGateway` implementation by defining a bean in the application context. Assign the `VaadinSessionInterceptor` to this gateway.

Note Note that there is an Event Hander that logs all `CardIssuedEvent`s to System Out. This logger also appends meta data assigned to that event. If you launch the application, you will not see much difference, that's because Axon is not instructed to copy this information from Command (where it is added by the interceptor) to the outgoing events.

Exercise 2 We need to instruct Axon to treat the Vaadin Session ID as correlation data. This way, it is passed on from command to event if the header is available in the meta data.

Define a bean of type `SimpleCorrelationDataProvider` that is instructed to copy the "`vaadin-session-id`" meta data element.

Since defining beans override defaults, we now don't have the `MessageOriginProvider` anymore. We can get it back by defining a bean that returns an instance of it.

Launch Launch the application and verify that the meta data is attached to each event.