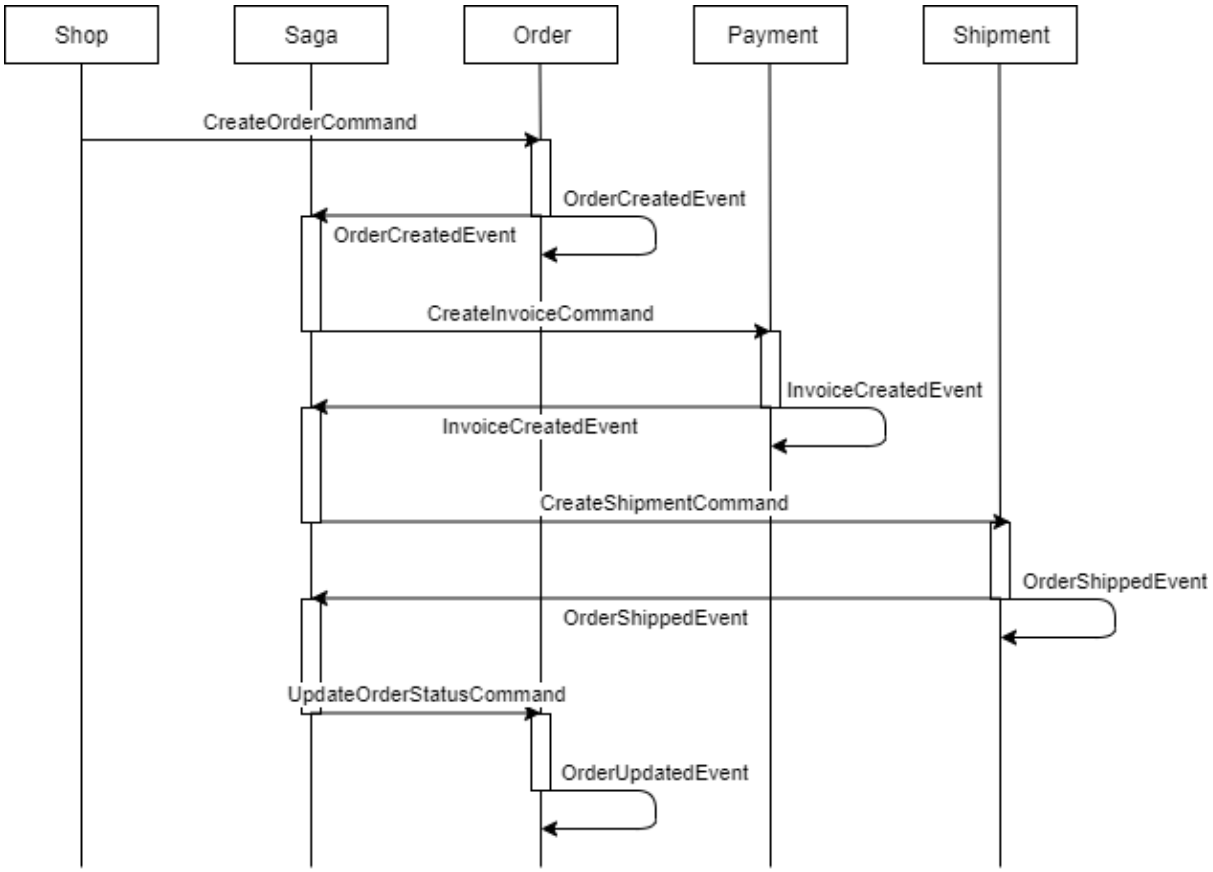
SAGA pattern lab with Axon

In this lab you are going to create a flow in which an order is placed, an invoice is created, and a shipment is done. You will place an order, after which the entire flow is triggered. An overview of this flow is given in the following image



You are going to create an Order management saga in the Order service using the Axon framework. This saga is responsible for the entire Order creation flow. You will also need to create the aggregates in the Order, Shipment and Payment service. To quickly start with the saga pattern using the Axon framework, a barebone for the Order service, Payment service and Shipment service is already given in the Git repository. All these services are Springboot applications.

The source code for this lab can be found in the AMIS github:

https://github.com/AMIS-Services/sig-saga-pattern

Clone/download the source code and load it into you favorite IDE.

If you have opened the source in your IDE, you will see 4 projects. The first is the core-apis, which contain the commands and the events classes, which are shared among the other 3 projects. The other 3 projects are the order-service, payment-service and the shipping-service. All three prjects are configured to run on their own port, 8080, 8081 and 8082 respectively.

When you open the pom of one of the three services, you will see the dependencies. Among these, you will see two dependencies:

```
<dependency>
   <groupId>org.axonframework</groupId>
   <artifactId>axon-spring-boot-starter</artifactId>
   <version>4.0.3</version>
</dependency>

<dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
   <scope>runtime</scope>
</dependency>
```

The first dependency is the spring starter for the Axon framework and contains all the Axon libraries needed to communicate with the Axon Server. The Axon Server itself will be run as a separate application and contains its own database. However, each service which communicates with the Axon Server also needs to store information. The services need to store which commands have been handled already and the association between a saga and the aggregates. For this purpose, the h2 database is used in this lab which persists its data to file, to ensure data is kept whenever we rebuild/restart a service. If you want to use the service in a production like environment, you would of course use a real database.

Order Service

The creation of the CreateOrderCommand is already implemented in the OrderCommandServiceImpl of the Order service, which is exposed on the REST endpoint /api/orders. What we must do is capture and handle this command in the Order service.For this open the OrderAggregate class. This class only contains a few fields. Let us make it a real aggregate which can handle the CreateOrderCommand.

First, set the annotation @Aggregate on the class and set the @AggregateIdentifier annotation on the orderId field.

- **@Aggregate** annotation tells Axon that this entity will be managed by Axon. Basically, this is like the @Entity annotation available with JPA.
- **@AggregateIdentifier** annotation is used for the identifying a particular instance of the Aggregate. In other words, this is like the JPA's @Id annotation.

Next, copy and paste the following code into this class:

```
@CommandHandler
public OrderAggregate(CreateOrderCommand createOrderCommand){
    LOG.info("Handle command with order id " + createOrderCommand.orderId);

    //
    // Some business logic
    //

    // Send the OrderCreated Event
    AggregateLifecycle.apply(new OrderCreatedEvent(createOrderCommand.orderId,
createOrderCommand.itemType,
            createOrderCommand.price, createOrderCommand.currency,
createOrderCommand.orderStatus));
}
```

This is a @CommandHandler annotated constructor, or differently put the 'command handling constructor'. This annotation tells the framework that the given constructor is capable of handling the CreateOrderCommand. Normally, @CommandHandler annotated functions contain the decision-making business logic. The `AggregateLifecycle.apply` method is used to publish an Event Message, in this case an OrderCreatedEvent.

Finally, copy and past the following code into the same class:

```
@EventSourcingHandler
protected void on(OrderCreatedEvent orderCreatedEvent){
    LOG.info("Handle event with order id " + orderCreatedEvent.orderId);
    this.orderId = orderCreatedEvent.orderId;
    this.itemType = ItemType.valueOf(orderCreatedEvent.itemType);
    this.price = orderCreatedEvent.price;
    this.currency = orderCreatedEvent.currency;
    this.orderStatus = orderCreatedEvent.orderStatus;
}
```

The @EventSourcingHandler is what tells the framework that the annotated function should be called when the Aggregate is 'sourced from its events'. The EventSourcingHandlers is the place where all the state changes happen.

Note that the Aggregate Identifier **must** be set in the @EventSourcingHandler of the very first Event published by the aggregate. Normaly, this is the creation event.

Open the CreateOrderCommand class. This class defines the payload of the command which we will be using to let an order be created. This command will be the first step of our entire flow.

1. Set the annotation @TargetAggregateIdentifier on the orderId field. This annotation tells the Axon framework which instance of the Aggregate type should handle the command. Since this command creates the Aggregate (it is passed as parameter to the OrderAggregate constructor) it is not needed to set the TargetAggregateIdentifier. However, it is recommended for consistency to annotate the identifier of all commands.

Open the OrderManagementSaga class. This class is the heart of our flow and already contains the beginning of the flow. In subsequent steps we will implement the other part of the flow. Note the **CommandGateway.** This object is used to dispatch commands.

We have to add three annotations to this class to get started:

@Saga annotation is used to let the Axon framework know that this is a Saga class. Set this on class level.

@SagaEventHandler is used to annotate that a method is an event handler. The `associactionProperty` defines the name of the property of the event that should be used to find the associated Sagas. Set the following annotation on the method handle(OrderCreatedEvent orderCreatedEvent):

```
@SagaEventHandler(associationProperty = "orderId")
```

@Saga In a saga, event handlers are annotated with `@SagaEventHandler`. If a specific event signifies the start of a transaction, add another annotation to that same method: `@StartSaga`. This annotation will create a new saga and invoke its event handler method when a matching event is published.

Give the method *handle(OrderCreatedEvent orderCreatedEvent)* a second annotation which will indicate that this method is the start of a saga: `@StartSaga`

If you look at the body of this method you will see that a unique payment id is generated and associated with this Saga. This means that in subsequent events we can correlate a payment with the generated id to this Saga.

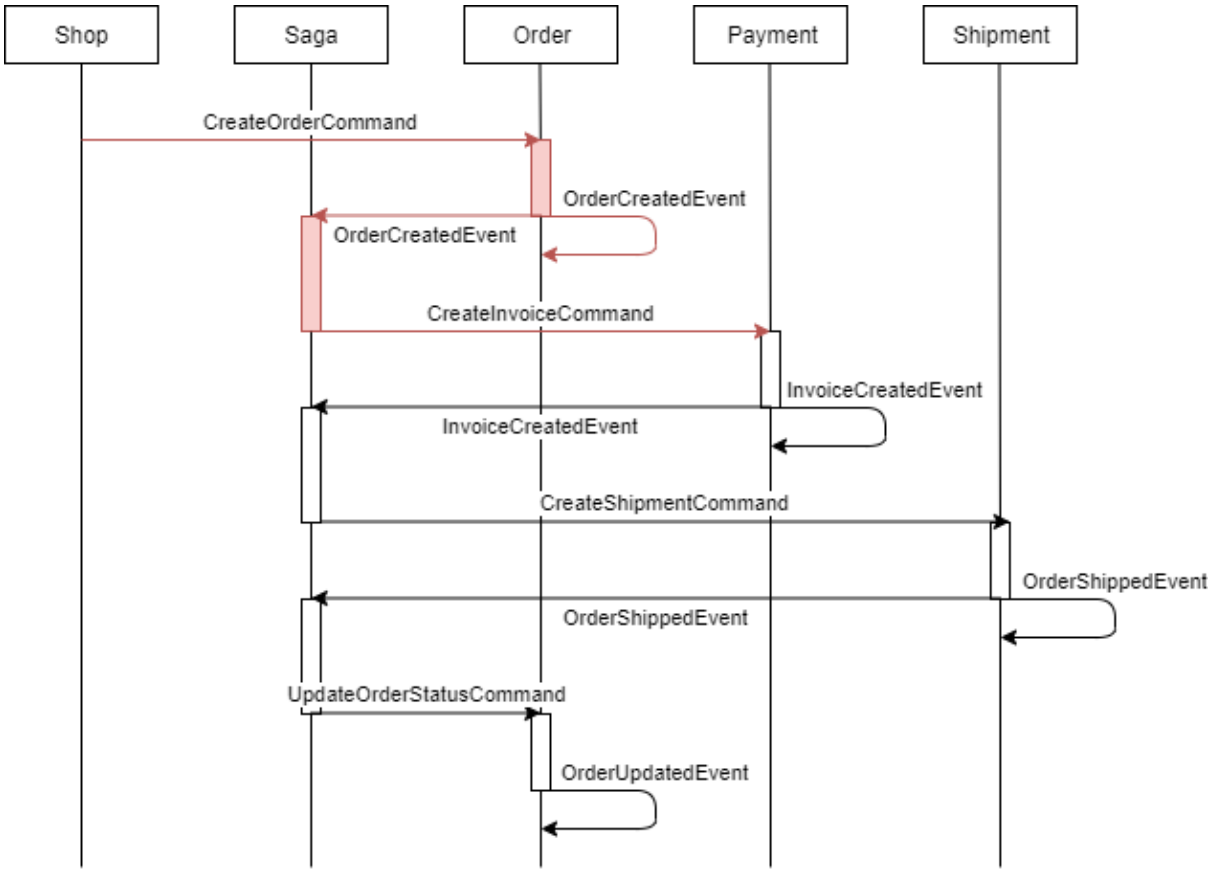
```
// Associate this Saga with the shipmentId
String paymentId = UUID.randomUUID().toString();
SagaLifecycle.associateWith("paymentId", paymentId);
```

Replace the line '// TODO send command' with the following code:

```
// Send the commands
CompletableFuture<String> result = commandGateway.send(new CreateInvoiceCommand(paymentId,
orderCreatedEvent.orderId, orderCreatedEvent.price));
```

This will create a command and send it using the command gateway. This command will be used to create a payment later on in the PaymentService.

Now that we have set up the first part of the flow, we can test it. For that we need an Axon Server. An overview of what we have created is shown in red in the following image:



Axon Server

You can run Axon using docker or by running a jar file. Choose the method whichever suits you the most.

Docker

```
docker run -d --name axonserver -p 8024:8024 -p 8124:8124 axoniq/axonserver
```

Jar

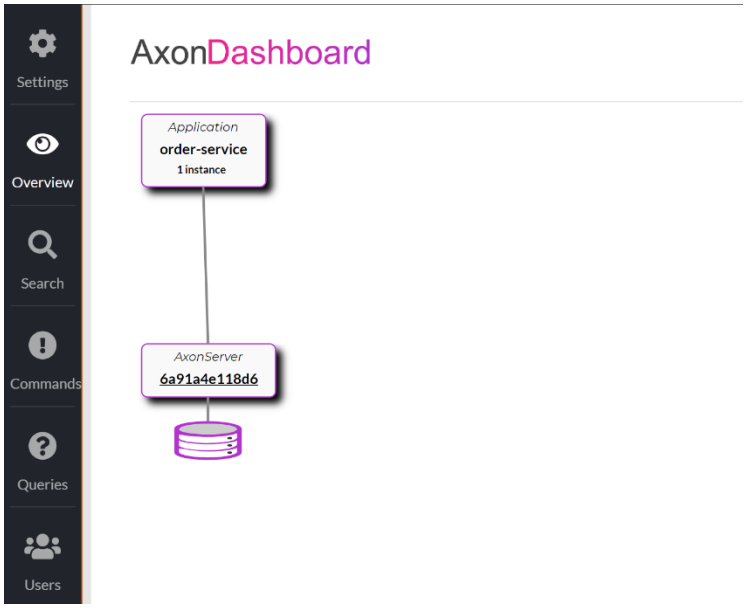
Download the jar file from

https://download.axoniq.io/axonserver/AxonServer.zip

and start the jar file using the command:

```
java -jar axonserver.jar
```

After the server has started, you can visit http://localhost:8024. The dashboard of the Axon server will be shown. Next, run the OrderService. Once the application has started successfully, you can see it in the Overview of the Axon Dashboard.



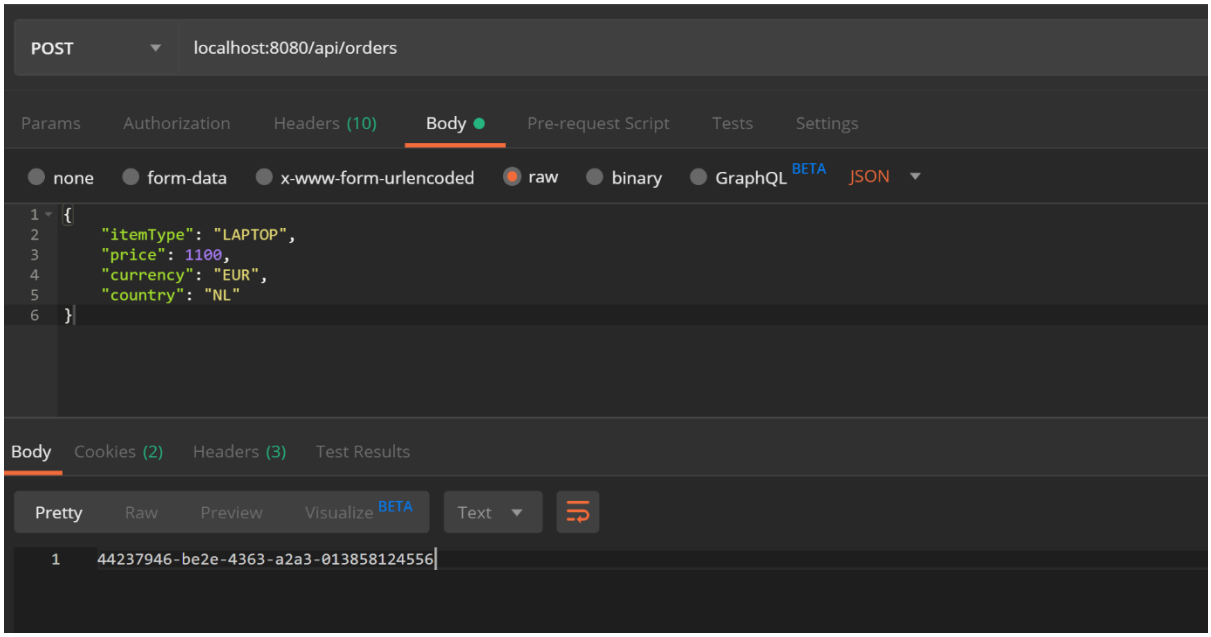
This page shows us all applications which have been registered to the Axon server. Later, when the PaymentService and ShippingService are running, you will see them in this dashboard next to the OrderService

Test the beginning of the flow

Open Postman and create a new POST request. Set the url to *localhost:8080/api/orders* and set the body to:

```
{
	"itemType": "LAPTOP",
	"price": 1100,
	"currency": "EUR"
}
```

Send the request.



A response with the order id of the newly created order is returned.

If you look in the log of the OrderService you will see a few lines:

```
2019-09-17 14:34:00.431  INFO 23104 --- [mmandReceiver-6] n.a.o.o.aggregates.OrderAggregate
: Handle command with order id 44237946-be2e-4363-a2a3-013858124556

2019-09-17 14:34:00.433  INFO 23104 --- [mmandReceiver-6] n.a.o.o.aggregates.OrderAggregate
: Handle event with order id 44237946-be2e-4363-a2a3-013858124556

2019-09-17 14:34:00.497  INFO 23104 --- [agaProcessor]-0] n.a.o.o.sagas.OrderManagementSaga
: Saga invoked with orderid 44237946-be2e-4363-a2a3-013858124556

2019-09-17 14:34:00.508  WARN 23104 --- [ault-executor-1]
o.a.c.gateway.DefaultCommandGateway      : Command
'nl.amis.ecommerce.commands.CreateInvoiceCommand' resulted in
org.axonframework.axonserver.connector.command.AxonServerRemoteCommandHandlingException(An
exception was thrown by the remote message handling component.)
```

As you can see, a command is handled by the OrderAggregate after which the corresponding event is handled. Next, a Saga is initiated with an order id corresponding to the response of our Postman request.

However, the last line gives a warning when the CreateInvoiceCommand is send. This is because no CommandHandler is defined for this command. So, to fix this error, let us create the rest of the flow.

Invoice Creation

In the OrderManagementSaga a CreateInvoiceCommand is sent. Since an invoice is another domain, the handling of the command is done in a different service, namely the PaymentService. This service is responsible for the invoice domain.

Open the PaymentService project and open the InvoiceAggregate. To make it an Aggregate class (like the OrderAggregate) apply the following steps to this class.

1. Add the annotations @Aggregate and @AggregateIdentifier
2. Create a second constructor with a CreateInvoiceCommand parameter. Annotate this constructor using the @CommandHandler annotation. Use AggregateLifecycle.apply to create an InvoiceCreatedEvent. As parameters, use the correct data from the CreateInvoiceCommand.
3. Give this aggregate an EventSourcingHandler which handles the InvoiceCreatedEvent. Creating this eventhandler will let us update the aggregate data to the correct state. So in this method you should set the state of this aggregate using the payload of the event. The invoiceStatus is not being sent in the payload of the event, so set it to InvoiceStatus.PAID since the event implies the payment is done.

So right now, we let the PaymentService handle the CreateInvoiceCommand. Also, the PaymentService sends an InvoiceCreatedEvent after the invoice command is handled to let the outside world know the invoice is successfully created.

InvoiceCreatedEvent SagaHandler

Let us go back to the OrderManagementSaga of the OrderService to let the saga act on the InvoiceCreatedEvent. You will see that this Saga already contains a function with a parameter of this event type.

1. Annotate this method with @SagaEventHandler and set the associationProperty to paymentId. Like the other handler, create a unique shippingId and associate the saga with this shippingId.
2. The payment is done and now we want to ship the order. Using the commandGateway, send a CreateShippingCommand.

You can rerun the OrderService and start the PaymentService to test if this part of the flow also works by sending a new request using Postman. If everything works fine, the log of the OrderService will look as follow:

```
2019-09-19 20:18:31.904  INFO 15204 --- [mmandReceiver-1] n.a.o.o.aggregates.OrderAggregate
: Handle command with order id 476bd0d0-87cc-4d18-9063-dd721c6e6310

2019-09-19 20:18:31.912  INFO 15204 --- [mmandReceiver-1] n.a.o.o.aggregates.OrderAggregate
: Handle event with order id 476bd0d0-87cc-4d18-9063-dd721c6e6310

2019-09-19 20:18:32.038  INFO 15204 --- [agaProcessor]-0] n.a.o.o.sagas.OrderManagementSaga
: Saga invoked with orderid 476bd0d0-87cc-4d18-9063-dd721c6e6310

2019-09-19 20:18:32.093  INFO 15204 --- [agaProcessor]-0] n.a.o.o.sagas.OrderManagementSaga
: Saga continued

2019-09-19 20:18:32.105  WARN 15204 --- [ault-executor-0]
o.a.c.gateway.DefaultCommandGateway      : Command
'nl.amis.ecommerce.commands.CreateShippingCommand' resulted in
org.axonframework.axonserver.connector.command.AxonServerRemoteCommandHandlingException(An
exception was thrown by the remote message handling component.)
```

The last line throws an exception just like we had before with the CreateInvoiceCommand. Once again this means that no handler is handling the CreateInvoiceCommand. Let us stitch the last part of the process together.

Create Shipping

Open the ShippingService. This service contains the ShippingAggregate in which we will handle the CreateShippingCommand. In the same way as we edited the InvoiceAggregate, let us apply the following changes to this class:

1. Add the annotations @Aggregate and @AggregateIdentifier
2. Create a second constructor with a CreateShippingCommand parameter. Annotate this constructor using the @CommandHandler annotation. Using AggregateLifecycle.apply create an OrderShippedEvent. As parameters, use the correct data from the CreateShippingCommand.
3. Give this aggregate an EventSourcingHandler which handles the OrderShippedEvent. In this method you should set the state of this aggregate using the payload of the event.

End of the Saga

We should handle the OrderShippedEvent in the OrderManagementSaga. Open the saga class. Create a new method named `handle` with as parameter the OrderShippedEvent. Also give this method the SagaEventHandler and set the associationProperty to the correct identifier of the event. In the body of this method, let us send a new UpdateOrderStatusCommand, with an orderstatus set to OrderStatus.SHIPPED. This command will allow us to update our order to be finished.

As the final step in our flow, we will handle the UpdateOrderStatusCommand. Open the OrderAggregate and create a new CommandHandler and a new EventSourcingHandler. Let the commandhandler send an OrderUpdatedEvent and set the state of the order aggregate to the payload of the event.

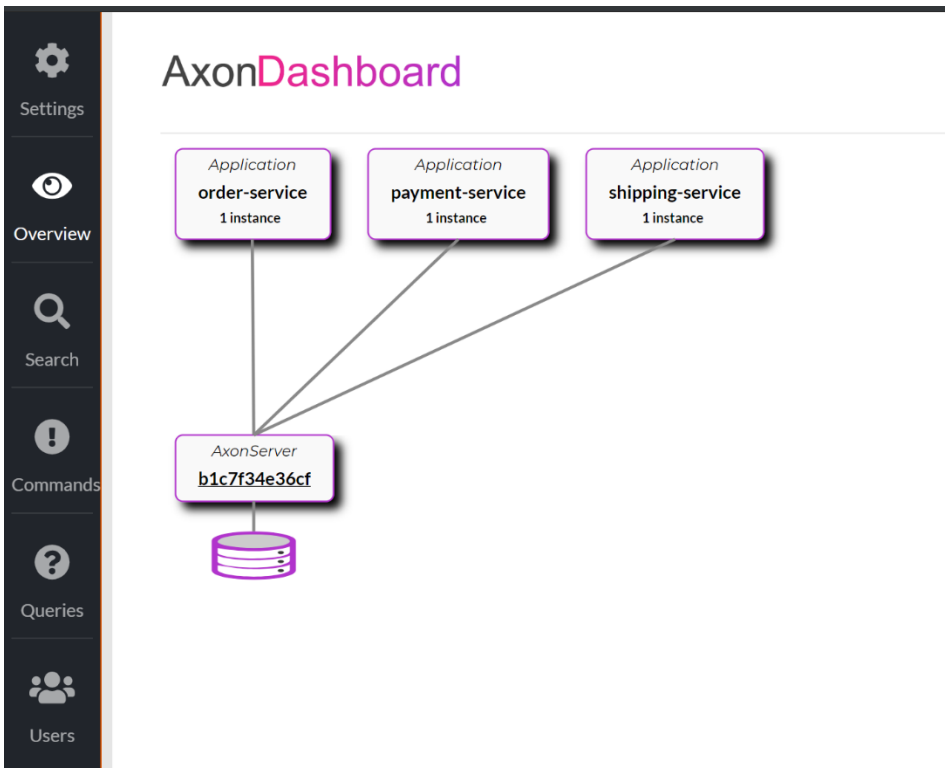
In the OrderManagementSaga, create the final SagaEventHandler, which listens to OrderUpdatedEvent and insert the following code into the body:

```
LOG.info("Saga ended");
SagaLifecycle.end();
```

This will finish the Saga whenever the an OrderUpdatedEvent is thrown for this saga.

Testing the entire flow

Start or restart all three services and wait for them to be started. If you look in the overview of the dashboard of the Axon Server, you will see three services connected to the Axon Server



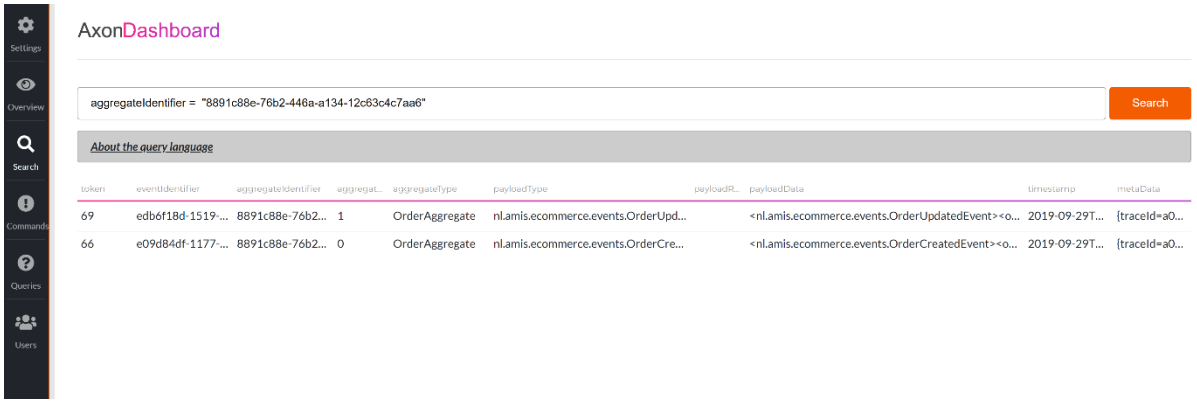
Using Postman, send a request to the OrderService. If everything goes well, you will see the following line in the logging of the OrderService:

```
n.a.o.o.sagas.OrderManagementSaga        : Saga ended
```

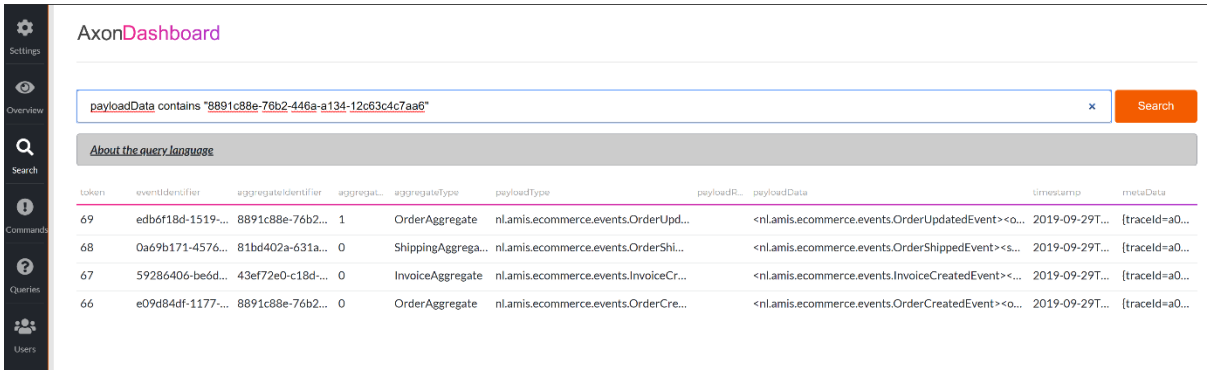
Let us view some more information in the Axon Server Dashboard. In the Overview, click on one of the three services. A more detailed screen of the service is shown. In here you can also view which Commands this service will handle. If you open the OrderService you can also view the OrderManagementSaga.

In the Commands tab you can view all commands and how many times a command has been handled by a service.

In the Search tab you can search for events. In here you will be able to view the data of all fired events. You can also search for events belonging to a specific order. Since the orderid of an order is also the aggregateidentifier of an Order object, we can use the this to search for our OrderAggregate. In the search bar search enter `aggregateIdentifier = "yourorderid"` where yourorderid is replaced with the orderid in the response of your Postman request.



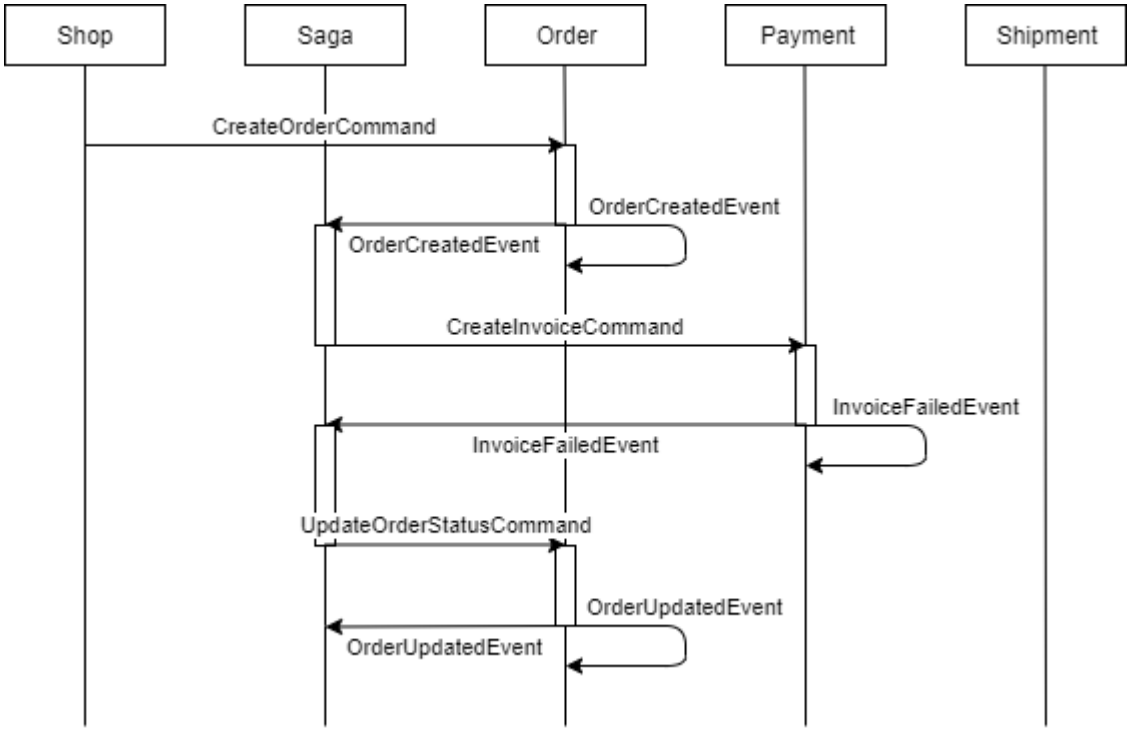
You can also search on the payload data of events. So, we can search all events which belong to our order:



This will give you a good overview of the flow of the order. You will notice that the aggregateSequenceNumber of the last event is 1 compared to 0 of the other events. This means the OrderAggregate is updated by an event and the number is the version of the OrderAggregate.

Payment Failed

We have created a happy flow for creating an order in which a payment is done, and a shipment has been carried out. However, in some scenarios it might be possible that a payment or shipment fails. After which all preceding actions must be undone. Let us work out the scenario in which a payment fails because the price has exceeded a predefined limit. Whenever a payment fails the order needs to be cancelled.



Open the InvoiceAggregate. In this class is the CommandHandler of the CreateInvoiceCommand. In here we will define the business logic of the creation of the payment. So, this will be the place to check if the payment can be fulfilled. Check if the price is higher than 1000 and send an InvoiceFailedEvent, otherwise send the InvoiceCreatedEvent. For convenience the following code checks if the price is bigger than 1000

```
createInvoiceCommand.price.compareTo(new BigDecimal(1000)) >= 0
```

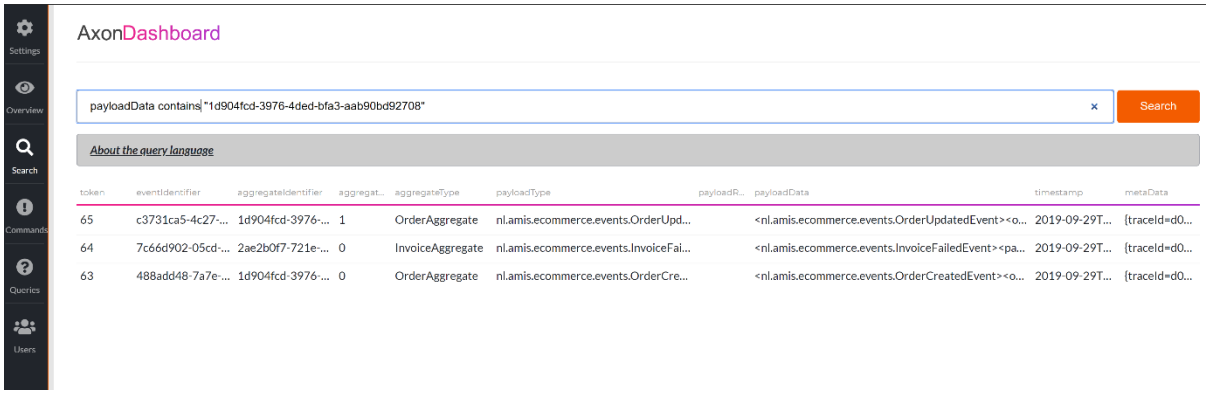
Add an EventSourcingHandler to this class, so that the InvoiceAggregate data can be updated using the InvoiceFailedEvent.

Also, this new InvoiceFailedEvent needs to be handled by the OrderManagementSaga. Open this class and add a new SagaEventHandler which handles this event correctly. Do not forget about the associationproperty. In this eventhandler let us send the command to update our order in which set the status to rejected.

Test Failed Payment

In Postman, send a request to localhost:8080/api/orders with a price higher than 1000. If you have added some logging to the eventhandlers, you can view the log of the services to check if you flow is correct. Otherwise, use the Search function in the Axon Dashboard.

Go to the Axon Dashboard and click on the Search tab. Search all events in which the payload contains your orderid. Check if the FailedEvent is sent and if the status of the OrderAggregate is set to REJECTED:



Bonus: Shipment Failed

Now you have seen how to create a happy flow and a flow in which the payment fails. However, failure of the shipment is also possible. Try to create the flow in which the payment succeeds but the shipment fails for some reason. Remember that you must reverse the payment as well. Let the saga handle this functionality when it receives some shipment fails event. Also, you will need to create new type of commands and new type of events for this scenario.